Département de formation doctorale en informatique **École doctorale IAE + M**
UFR STMIA

Compilation de règles de réécriture et de stratégies non-déterministes

THÈSE

présentée et soutenue publiquement le 22 juin 1999

pour l'obtention du

Doctorat de l'université Henri Poincaré – Nancy 1

(spécialité informatique)

par

Pierre-Etienne Moreau

Composition du jury

Président :	Yves Caseau	Directeur de recherche, Bouygues, France
Rapporteurs :	Guy Cousineau	Professeur, Université Denis Diderot - Paris VII, France
	Michael J. O'Donnell	Professeur, Université de Chicago, USA
	Karl Tombre	Professeur, École des Mines de Nancy, France
Examinateurs :	Alexander Bockmayr	Professeur, Université Henri Poincaré - Nancy 1, France
	Hélène Kirchner	Directeur de recherche, CNRS, France

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

Mis en page avec la classe thloria.

À mon père

Remerciements

Le développement de cette thèse m'a particulièrement occupé ces dernières années, mais les motivations sous-jacentes ont mijoté dans mon esprit pendant plus de dix ans. C'est en commençant à imaginer des algorithmes pour énumérer des nombres premiers que certains signes laissaient prévoir mon intérêt pour l'expressivité et l'efficacité des langages de programmation. C'est après avoir réalisé un premier interpréteur capable de tracer et d'étudier une fonction que j'ai commencé à m'intéresser clairement au « calcul symbolique » et aux notions de structures de données telles que les arbres et les piles. De manière presque « magique », la réunion de ces notions permettait de représenter des fonctions et de calculer leur dérivée ou des valeurs en certains points.

Par la suite, un projet scolaire m'a amené à concevoir et réaliser un compilateur pour un sous ensemble du langage Pascal. Je dois avouer que l'idée d'analyser la structure du programme cible pour déterminer à l'avance de quelle façon allouer les registres, m'a particulièrement enchantée. Je pense que cette période date approximativement mon attrait pour la compilation. Je tiens à remercier tout spécialement Karol Proch, qui était à l'initiative de ce projet.

Je remercie particulièrement Nicolas Trotignon et Eugen Calapodescu, deux grands amis qui m'ont aidé à façonner mes idées de milliers de façons différentes ; leurs influences et leurs idées se retrouvent forcement dans cette thèse.

Je n'aurais pas pu entreprendre cette thèse si Marian Vittek n'avait pas créé l'environnement ELAN. Quand j'ai été en quête de conseils de petite ou de grande portée, je me suis souvent adressé à Marian, qui connaît le langage et le compilateur comme s'il l'avait fait. C'est aussi à lui que je dois une grande partie des difficultés rencontrées.

Peter Borovanský peut être considéré comme le deuxième créateur d'ELAN. Il m'a continuellement aidé à avancer en améliorant sans cesse le langage et son interpréteur. C'est certainement grâce à Peter que des applications complexes ont pu voir le jour et sans lui, le compilateur aurait été terminé bien plus tôt, mais cela n'aurait été qu'un prototype.

Je ne remercierai jamais assez Eric Domenjoud de m'avoir un jour posé la question suivante : « que fais-tu au juste ? je n'ai toujours pas compris le sujet de ta thèse ». Son esprit critique et curieux m'a amené à travailler sur un sujet profondément passionnant. Nous avons aussi passé de nombreuses heures à refaire le monde et je l'en remercie.

Thomas Genet, avec qui j'ai partagé mon premier bureau, m'a supporté et soutenu pendant tout le développement du compilateur. Je dois beaucoup à sa bonne humeur permanente.

Merci à Christophe Ringeissen et Laurent Vigneron pour leurs nombreux commentaires et conseils relatifs au manuscrit et aux transparents de la soutenance. Ils ont activement contribué à améliorer la qualité de l'ensemble.

Merci à Horatiu Cirstea, Hubert Dubois, Christelle Scharff et les autres pour m'avoir aidé à accroître la stabilité du compilateur. Chaque disfonctionnement signalé a pu paraître minime, mais cela a été d'une grande aide.

Je n'aurais sûrement pas pu aboutir à un tel document sans l'existence d'outils tels que TEX, LATEX et MetaPost. Je tiens à remercier fortement Denis Roegel pour sa disponibilité permanente et la qualité de son travail. Bien plus qu'un simple « gourou », je suis persuadé que sa persévérance et son perfectionnisme ont influencé ma façon d'aborder un problème et par conséquent cette thèse.

Je n'aurais certainement pas eu l'ambition ni la volonté de développer des algorithmes aussi pointus sans la « rivalité » de Steven Eker. Bien que situé à plusieurs milliers de kilomètres de

Nancy, les nombreux échanges de *mails* nous ont entraîné dans une compétition sans fin qui a permis d'améliorer indiscutablement les algorithmes de filtrage et de normalisation modulo AC.

Le séjour à Nancy de Bernhard Gramlich a été d'une grande richesse pour son entourage. Son intérêt constant pour le travail des autres et les nombreuses discussions passées autour d'un café m'ont sans nul doute ouvert les yeux et aidé à faire des choix fondamentaux. Je tiens ainsi à le remercier particulièrement.

Je tiens à remercier Paul Klint et Mark van den Brand pour m'avoir invité 1 mois au CWI et m'avoir initié aux secrets d'ASF+SDF. Les nombreux échanges scientifiques ont largement influencé ma façon de voir et concevoir un environnement de spécification. Les travaux de Mark et de Pieter Olivier sur la compilation de systèmes de réécriture m'ont eux aussi influencé et motivé.

On dit souvent que le hasard fait bien les choses et j'ai pu le vérifier : c'est dans une période de doute que Kostis Sagonas et Bart Demoen se sont intéressés à mes travaux. Leur intérêt et leurs encouragements m'ont été d'une très grande aide et je les remercie particulièrement.

Je remercie Brigitte et Jacques Jaray pour avoir toujours cru en moi. C'est en particulier grâce à Brigitte que je suis venu à Nancy et c'est aussi elle qui m'a incité à faire un premier stage dans l'équipe Prothéo. Jacques a accepté d'être mon tuteur et il m'a aidé à faire mes premiers pas dans le monde de l'enseignement. J'ai particulièrement apprécié sa disponibilité, sa confiance et ses conseils. Je les remercie grandement tous les deux.

J'en profite aussi pour remercier tous mes élèves de l'École de Mines de Nancy, de l'Université Henri Poincaré — Nancy 1 et de l'Université Nancy 2 pour avoir rendu passionnant mon travail d'enseignant.

Merci à Michaël Rusinowich et à Paul Zimmermann pour leur intérêt permanent et leur regard « extérieur ».

Bien que ne dirigeant pas ma thèse, je pense pouvoir dire que Claude Kirchner a « co-encadré » une grande partie de mes travaux. Par son implication dans le projet ELAN, par sa grande confiance et par sa constante disponibilité, il m'a en permanence aidé à faire des choix difficiles et à croire en mes idées. Sa gentillesse, son soutien et sa passion pour la recherche ont sans aucun doute contribué à cette thèse. Je lui en suis extrêmement reconnaissant.

À une période de l'année où il était très occupé, Guy Cousineau m'a honoré en acceptant d'être rapporteur de cette thèse. Son ouverture et sa lecture attentive m'ont fait découvrir un état d'esprit d'une grande valeur. J'ai été particulièrement touché par ses commentaires sur le manuscrit et par ses questions au cours de la soutenance. Je tiens à le remercier tout particulièrement pour sa disponibilité et sa confiance.

Michael J. O'Donnell m'a fait l'honneur d'être rapporteur de cette thèse et n'a pas hésité à venir spécialement de Chicago pour participer à la soutenance. Les discussions que nous avons eues ainsi que ses remarques sur le document m'ont été très précieuses. Je tiens aussi à le remercier d'avoir accepté de lire tout le manuscrit en français.

Si je dois remercier quelqu'un pour m'avoir donné envie de poursuivre mes études au Loria, c'est bien Karl Tombre, qui m'a accueilli en stage voila bientôt 5 ans. Bien que travaillant dans un autre domaine, il a accepté d'être rapporteur de cette thèse. J'ai ainsi pu profiter de ses remarques précieuses sur le document, de sa rigueur et de sa « vision » de l'informatique. Sans le savoir, il m'a continuellement incité à clarifier mes explications en étant mon « lecteur imaginaire » tout au long de la rédaction. Je tiens à le remercier amicalement.

Je tiens à remercier Alexander Bockmayr pour avoir accepté d'examiner ce document et de participer à mon jury. Par ses questions et ses remarques il m'a témoigné un grand intérêt pour les travaux effectués.

C'est un euphémisme de dire qu'Yves Caseau est très occupé. Il a pourtant immédiatement accepté de me consacrer du temps en étudiant mes travaux et ce document. Yves Caseau m'a fait l'honneur de présider mon jury de thèse et par ses « vraies » questions, il m'a communiqué son attrait pour les problèmes complexes et son intérêt pour le travail réalisé. L'idée de travailler avec lui me motive particulièrement.

Je tiens à remercier tout spécialement Hélène Kirchner, ma directrice de thèse, pour m'avoir aidé et guidé tout au long de la préparation de cette thèse. Du premier jour au dernier jour, Hélène a toujours été présente pour discuter, étudier une proposition, remettre en cause un choix, proposer une alternative et s'intéresser à mes idées (parfois peu claires). Par sa compétence, sa confiance et sa sympathie, elle m'a toujours aidé à transformer en réussite les situations d'échec. C'est bien à elle que je dois ma passion pour la recherche et je tiens à la remercier très sincèrement.

Grâce à son livre, Douglas Hofstadter a réveillé en moi un intérêt pour la rédaction, l'écriture et la présentation d'idées « complexes ». Je lui en suis très reconnaissant.

Merci à tous mes amis du laboratoire, de Nancy, de Paris, de Strasbourg, de Forbach, de Lyon, de Libourne et d'ailleurs.

J'ai essayé de me souvenir de tous ceux qui ont contribué à cette thèse, mais je n'ai sans aucun doute pas réussi à les citer tous.

Je dois plus à ma famille qu'à toute autre personne. Elle m'a guidé, encouragé et soutenu. Et surtout, elle a toujours cru en moi. C'est à elle que cette thèse est dédiée.

Pierre-Etienne Moreau
Nancy
Juillet 1999

Sommaire

Avant-propos xiii

Introduction 1

I Environnement de spécification 9

1 Langage de spécification ELAN 11

1.1 Grammaire et signature . . . 11
1.2 Termes et règles de réécriture conditionnelles . . . 13
1.3 Stratégies d'application . . . 15
1.4 Règles et stratégies . . . 16
1.5 Opérateurs Associatifs et Commutatifs . . . 21
1.6 Modularité . . . 23

2 Outils pour spécifier et programmer 27

2.1 Bibliothèque . . . 27
2.2 Parseur . . . 29
2.3 Interpréteur . . . 30
2.4 Compilateur . . . 32
2.5 Comparaison avec d'autres environnements de spécification . . . 33

3 Plateforme de prototypage 39

3.1 Format d'échange . . . 40
3.2 Création d'outils . . . 43
3.3 Système ouvert . . . 45
3.4 Vers une nouvelle architecture . . . 46
3.5 Synthèse . . . 49

II Compilation de la réécriture **51**

4 Méta-conception **53**
4.1 Interpréteur, Compilateur et Machine abstraite . . . 53
4.2 Pourquoi choisir un compilateur . . . 56
4.3 Compilation de la réécriture . . . 58

5 Compilation du filtrage syntaxique **61**
5.1 Termes vus comme des chaînes de symboles . . . 62
5.2 Automate de filtrage . . . 63
5.3 Clôtures d'un ensemble de motifs . . . 66
5.4 Clôture « réduite » d'un ensemble de motifs . . . 70
5.5 Automate de filtrage à mémoire . . . 72
5.6 Automate de filtrage avec jumpNode . . . 75
5.7 Comparaison des différentes approches . . . 78

6 Compilation du filtrage associatif-commutatif **81**
6.1 Termes en forme canonique . . . 82
6.2 Approche one-to-one . . . 83
6.3 Approche many-to-one . . . 84
6.4 Classes de motifs . . . 88
6.5 Spécialisation utilisant une structure compacte . . . 88
6.6 Raffinement glouton . . . 93
6.7 Calcul des substitutions . . . 94
6.8 Extension à l'ensemble des motifs . . . 95
6.9 Synthèse . . . 97

7 Gestion du non-déterminisme **99**
7.1 Introduction . . . 100
7.2 Basic choice point primitives . . . 101
7.3 Known choice point implementations . . . 102
7.4 New choice point management . . . 103
7.5 Imperative programming with backtracking . . . 109
7.6 Concluding Remarks . . . 111

8 Compilation des règles et des stratégies **113**
8.1 Tour d'horizon . . . 113
8.2 Solution retenue pour ELAN . . . 115

8.3 Compilation du filtrage et de la sélection des règles . . . 116
8.4 Compilation des évaluations locales . . . 118
8.5 Construction du terme réduit . . . 124
8.6 Compilation des stratégies . . . 127

9 Analyse du déterminisme **131**
9.1 Stratégies primitives . . . 132
9.2 Classification du déterminisme . . . 132
9.3 Inférence de la classe de déterminisme . . . 134
9.4 Impact de l'analyse du déterminisme . . . 136
9.5 Résultats expérimentaux . . . 138

III Implantation d'un compilateur **141**

10 Architecture logicielle **143**
10.1 Compilation modulaire et compilation séparée . . . 143
10.2 Organisation du compilateur . . . 147
10.3 Fonctionnement du compilateur . . . 150

11 Support d'exécution **153**
11.1 Structures de données . . . 153
11.2 Opérations internes . . . 155
11.3 Sortes et opérations prédéfinies . . . 156
11.4 Gestion de la mémoire . . . 158
11.5 Synthèse . . . 163

12 Expériences pratiques **165**
12.1 Estimation du degré de compilation . . . 166
12.2 Évaluation des performances . . . 172
12.3 Coût du filtrage AC . . . 176
12.4 Comparaison avec d'autres implantations . . . 178

Conclusion **183**

Annexes **193**

A Programmes utilisés pour effectuer les expérimentations **193**
A.1 Brute . . . 193
A.2 Caml . . . 204

A.3 Cime 205
A.4 Elan 211
A.5 Maude, Obj 221
A.6 Otter 229
A.7 Redux 231
A.8 Rrl 233

Bibliographie **235**

Index **245**

Résumé

Résumé Les techniques de réécriture ont été développées depuis les années 1970 et appliquées en particulier au prototypage des spécifications formelles algébriques et à la démonstration de propriétés liées à la vérification de programmes.

ELAN est un système qui permet de spécifier et d'exécuter des résolveurs de contraintes, des démonstrateurs et plus généralement tout processus décrit par des règles de transformation. Il possède des opérateurs associatifs-commutatifs (AC) et un langage de stratégies qui permettent une gestion fine de l'exploration d'un arbre de recherche et une manipulation aisée d'opérateurs mathématiques tels que les connecteurs booléens, les opérateurs arithmétiques ou les opérateurs de composition parallèle par exemple.

Ces deux notions améliorent grandement l'expressivité du langage mais introduisent un double non-déterminisme lié à la possibilité d'appliquer plusieurs règles, de différentes façons, sur un terme donné. Cela rend difficile et généralement peu efficace leur implantation.

L'objectif principal de cette thèse est d'étudier des techniques de compilation qui améliorent l'efficacité de ce type de langages. Nous proposons un nouvel algorithme, à base d'automates déterministes, pour compiler efficacement le filtrage syntaxique. Nous définissons ensuite différentes classes de règles pour lesquelles nous proposons un algorithme efficace de filtrage AC. Cet algorithme utilise une structure de donnée compacte et les automates définis précédemment, ce qui améliore considérablement les performances du processus de normalisation dans son ensemble.

L'étude du langage de stratégies conduit à implanter des primitives originales de gestion du *backtracking* et à définir un algorithme d'analyse du déterminisme permettant de réduire leur usage et d'améliorer encore les performances, tout en réduisant l'espace mémoire nécessaire. Enfin, l'implantation des méthodes proposées a donné lieu à l'élaboration de nombreuses optimisations théoriques et techniques qui peuvent être largement réutilisées pour implanter d'autres langages de programmation par réécriture. Cette thèse présente les algorithmes et leur évaluation, l'architecture et le fonctionnement du compilateur, ainsi qu'une proposition d'environnement de spécification, fondée sur l'utilisation d'un format intermédiaire.

Mots-clés: Compilation, système de réécriture, stratégie, filtrage associatif-commutatif, non-déterminisme.

Abstract

Abstract Rewriting techniques are developed since 1970 and applied, in particular, to prototyping formal algebraic specifications and to proving properties related to program verification.

ELAN is a system designed to specify and execute theorem provers, constraints solvers and more generally, any process described by transformation rules. It supports a strategy language useful to make a precise exploration of a search space. It also supports associative-commutative (AC) operators that make easier the study of mathematical operators such as boolean connectors, arithmetic operators or parallel composition operators, for example.

Those two notions greatly improve the expressivity but introduce a double non-determinism that generally makes difficult and inefficient their implementation.

The main purpose of this thesis is to study compilation techniques that improve the efficiency of this kind of language. We propose a new algorithm, based on deterministic automata, to efficiently compile the syntactic matching process. Then, we define several classes of patterns and a compact data structure in order to improve the efficiency of the AC matching algorithm. Automata described above are used by the algorithm, and the results show an impressive improvement of the whole normalisation process.

The study of the strategy language leads us to design two new backtracking primitives to handle nondeterminism. Moreover we define a determinism analysis algorithm in order to reduce their use, further improve the efficiency, and reduce the needed memory usage. While implementing the proposed methods, a lot of theoretical and technical optimisations have been designed, and can be reused to implement other rewriting based languages. This thesis describes the algorithms and their evaluation, the architecture and the design of the compiler, as well as a proposal for a specification environment, based on the existence of an intermediate format.

Keywords: Compilation, rewriting system, strategy, associative and commutative matching, nondeterminism.

Avant-propos

« Faire une thèse » fait peut-être partie des expressions qui ne vous impressionnent plus autant qu'il y a quelques années, au même titre que « je reviens de San Francisco » ou « je pars faire un exposé à Hawaï ». Même sans revenir de San Francisco, cette expression laisse généralement indifférent l'étudiant qui prépare sa thèse. Mais ce n'est pas toujours le cas des personnes rencontrées ici ou là. Il m'est ainsi arrivé, au cours de discussions, de repas ou de fêtes, de rencontrer des personnes, des amis ou des proches réellement intrigués par cette expérience qu'est la préparation d'une thèse. Comment dans ce cas ne pas échapper à la question « je n'y comprendrai sûrement rien, mais quel est le sujet de ta thèse ? ».

C'est évidemment une preuve d'intérêt que de se voir questionné sur ses recherches, mais ce n'est pas sans rappeler le merveilleux film d'Alain Resnay : « On connaît la chanson ». Qui n'a pas souri en écoutant la pauvre Camille (Agnès Jaoui) nous expliquer qu'elle s'intéressait aux chevaliers paysans de l'an 1000 au Lac de Paladru ?

C'est principalement pour éviter ce petit sourire et l'immanquable réponse qui l'accompagne : « Ah oui, ce doit être réellement passionnant », que je n'ai jamais osé dire que je travaillais principalement sur la compilation efficace du filtrage associatif-commutatif en présence de stratégies non-déterministes. J'ai toujours préféré contourner le problème en répondant : « je veux bien te répondre, mais il faut que tu me donnes au moins cinq minutes ». Écoutons maintenant à quoi ressemblait la conversation avec les plus courageux.

Le courageux : J'ai cru comprendre que tu faisais de l'informatique. Avec Internet ...

Moi : Heu oui, disons que de très très loin, je travaille dans une branche liée à l'Intelligence Artificielle, et plus précisément, dans un secteur relié aux preuves mathématiques.

Le littéraire : C'est donc de l'informatique, mais aussi des mathématiques que tu fais. Ça, ce n'est pas pour moi.

Moi : En fait, je travaille sur les preuves de propriétés de programmes, mais pour te donner un idée plus précise, tu peux penser aux satellites qu'on envoie dans l'espace.

Le bricoleur : Oui, j'imagine qu'il y a un tas d'électronique là dedans.

Moi : En effet. Il y a aussi beaucoup de programmes, et lorsqu'on pose une question au satellite, on a envie qu'il nous réponde assez rapidement.

L'intéressé : ???

Moi : Imagine par exemple qu'on veuille modifier la trajectoire d'un satellite. Dans ce cas, on lui demande où il se trouve, ou des informations sur sa vitesse par exemple. Mais s'il nous répond le lendemain, ou si le programme part dans une boucle infinie et qu'il ne nous répond jamais, on est embêté.

Le secouriste : D'autant plus qu'on ne peut pas le ramener facilement sur Terre pour le réparer. m

Moi : C'est principalement pour cela, qu'avant de l'envoyer dans l'espace on a envie de « certifier » son électronique et ses programmes. On a par exemple envie de « prouver » que les programmes sont corrects.

Le lecteur du « Monde » :
On dit qu'Ariane 5 a explosé parce qu'il y avait une erreur dans un programme.

Moi : Il y a du vrai, et c'est relié de très loin à ce que je fais. Pour revenir aux satellites, on a envie qu'ils répondent à nos questions de façon cohérente et dans un temps relativement court, moins de 10 minutes par exemple. C'est ce qu'on appelle une *propriété* d'un programme. Imagine que tu appuies sur la pédale de frein de ta voiture et qu'elle ne freine effectivement que 30 secondes plus tard.

Le commercial : Ce serait embêtant.

Moi : Ne va pas imaginer que je travaille sur l'envoi de satellites ou la conception d'un système de freinage. Je travaille en amont pour essayer de prouver que les programmes, des satellites ou des voitures par exemple, sont corrects. Mais ce n'est pas moi qui fais les preuves. Dans l'équipe il y a des chercheurs qui font des programmes pour faire les preuves automatiquement.

Le réconfortant :
Ce ne doit pas être facile tout ça.

Moi : Pas tellement, mais sinon, ça va ? Tu me suis encore ? Maintenant, on monte d'un niveau : en réalité, je fais des outils pour les chercheurs qui fabriquent ces programmes.
(*après généralement un ou deux signes d'étonnement*)

L'attentif : Quel genre d'outil ?

Moi : Ces outils, ce sont des programmes, et dans notre sous-groupe de recherche, on essaie de fabriquer un nouveau langage pour écrire plus facilement ces outils.
(*petite pause*)

Moi : Pour résumer : il y a des démonstrateurs automatiques qui sont utilisés pour prouver que d'autres programmes sont corrects. Et nous, on travaille sur un langage qui nous permet de prouver plus facilement que les démonstrateurs, *eux-mêmes*, sont corrects.

Le logicien : C'est vrai que si le démonstrateur est faux, il risque de prouver n'importe quoi, et on ne serait pas plus avancé.

Moi : Je travaille ainsi sur l'élaboration d'un nouveau langage, mais aussi sur les outils qui permettent d'exécuter les programmes écrits dans ce langage. Tu vois, on monte encore d'un niveau.
(*petite pause*)

Moi : Un des objectifs de ma thèse, c'est de faire en sorte que les programmes, écrits dans ce nouveau langage, aillent le plus vite possible.

Le curieux : Et c'est quoi le langage que vous inventez ?

Moi : Il s'appelle ELAN, mais il n'est pas encore connu et ne le sera sûrement jamais. Enfin, peut-être que dans quelques années, 7 ans ou 14 ans, de nouveaux langages s'inspireront, de près ou de loin, de ce qu'on a fait ...

Introduction

En janvier 1937 paraissait l'article d'Alan Turing sur les « nombres calculables », ce qui date approximativement l'apparition de la notion *système formel*. Il semble cependant, d'après (Hodges 1988, Hofstadter 1985), que certaines des idées de Gödel et Turing aient été anticipées dès le début des années 1920 par le logicien polono-américain Emil Post qui enseignait au City College de New York.

Descendant des « systèmes de production de Post », la notion de programmation fonctionnelle s'est largement développée dans les années 1960, suite aux travaux de John McCarthy sur le langage Lisp. C'est aux alentours de l'année 1975 que la notion de *programmation par équations* ou par *règles de réécriture* est effectivement apparue, suite aux travaux de Joseph A. Goguen et de Michael J. O'Donnell, menant aux développements des premiers interpréteurs de spécifications exprimées avec des règles de réécritures. Bien que relativement proches au départ, les deux projets ont suivi des voies radicalement différentes.

Il existe évidemment de nombreux autres travaux reliés aux notions de *règles de réécriture* et de *programmation par équations*, parmi lesquelles on peut citer les démonstrateurs automatiques que sont C*i*ME (Marché 1996), daTac (Vigneron 1998), Larch Prover (Guttag, Horning, Garland, Jones, Modet et Wing 1993), Otter (McCune 1994), ReDuX (Bündgen 1993), Reve (Lescanne 1983, Forgaard et Guttag 1984), RRL (Kapur et Zhang 1988) et Spike (Bouhoula, Kounalis et Rusinowitch 1992), les langages fonctionnels de la famille ML (Cousineau, Paulson, Huet, Milner, Gordon et Wadsworth 1985) et Caml (Weis et Leroy 1993, Cousineau et Mauny 1995, Leroy et Mauny 1993, Leroy 1995), et les langages de réécriture de graphes tels que Clean (Brus, van Eskelen, van Leer et Plasmeijer 1986). Dans le cadre de cette thèse nous nous intéressons particulièrement aux outils qui utilisent des règles de réécriture, et plus précisément aux langages de programmation dont le paradigme de calcul principal est celui de la logique de réécriture. La plupart des outils cités précédemment utilisent la réécriture comme technique interne de résolution, mais afin de mieux situer nos travaux, nous les comparons essentiellement avec ceux des projets ASF+SDF, CafeOBJ, EQI, Maude et OBJ, simplement parce que ces systèmes sont exclusivement fondés sur la logique de réécriture et parce qu'ils sont les plus proches d'ELAN. La figure 1 retrace sommairement l'évolution des principaux langages de programmation fondés sur la logique de réécriture.

Le projet « Equational Logic Programming » de Michael J. O'Donnell s'est particulièrement intéressé aux propriétés liées à l'évaluation des systèmes équationnels : le modèle choisi permet en particulier d'exploiter la notion de stratégie « paresseuse », ce qui permet de retarder au maximum l'évaluation des arguments d'une fonction au cours des étapes de déduction. Ces travaux ont conduit Robert Strandh à étudier comment implanter efficacement les langages à base de règles de réécriture. En 1986, il proposa le premier compilateur pour un tel langage et décrivit son fonctionnement dans sa thèse (Strandh 1988). Au cours de sa thèse, David J. Sherman (1994) développa de nouvelles techniques de compilation permettant d'améliorer les performances du compilateur : la spécification initiale est dans un premier temps transformée

par évaluation partielle, et des techniques de partage sont utilisées à l'exécution pour minimiser le nombre de symboles à construire et mettre en facteur des séquences de calcul redondantes.

Le projet « OBJ » de Joseph A. Goguen s'est quant à lui particulièrement concentré sur le formalisme de spécification et sur l'expressivité du langage développé : le langage OBJ permet ainsi de définir des inclusions de sortes, des réécritures modulo les axiomes d'associativité et de commutativité ainsi que des expressions de modules paramétrés. Les premières versions du langage furent développées principalement par Futatsugi, Goguen, Jouannaud et Meseguer, et c'est en 1987, que la version OBJ-3 (Goguen, Kirchner, Kirchner, Mégrelis, Meseguer et Winkler 1987) fut présentée. Ces travaux sur la dernière version d'OBJ ont certainement influencé le projet ELAN, démarré par Claude et Hélène Kirchner en 1990. Un autre projet, Maude, fut parallèlement démarré par José Meseguer. On pourrait croire que l'histoire se répète en voyant à nouveau ces deux projets s'orienter vers des voies différentes.

Le projet Maude de José Meseguer s'est orienté vers la définition d'un formalisme plus riche que celui d'OBJ-3, en intégrant la notion de réflexivité, le paradigme de programmation objet et les notions de réécriture modulo les théories associatives et leurs extensions aux mélanges avec d'autres axiomes comme l'idempotence ($f(x,x) = x$) et l'élément neutre ($f(x,e) = x$). Le premier interpréteur Maude, développé par Steven Eker, fut présenté en 1996 et diffusé en 1998.

Le projet ELAN, démarré par Claude et Hélène Kirchner, s'est quant à lui orienté vers l'aspect opérationnel de la réécriture en introduisant, pour la première fois, la notion de *stratégie* définie par l'utilisateur. De telles stratégies permettent, par exemple, d'explorer un espace de recherche en contrôlant finement l'ordre d'application des règles de réécriture. Au cours de sa thèse (1994), Marian Vittek proposa et implanta le premier environnement de programmation pour ELAN.

Un troisième projet : « ASF+SDF », fut quant à lui démarré dans les années 1980 par Jan Heering et Paul Klint. L'objectif était de définir un environnement de programmation générique permettant d'éditer, d'exécuter et de déboguer des programmes écrits dans un langage spécifié par une grammaire. Suite à un séjour en France et après avoir étudié le système Mentor de l'INRIA, Paul Klint utilisa l'*Equation Interpreter* (EQI) de O'Donnell pour le comparer à d'autres façons d'implanter un langage à base de réécriture. C'est en 1989 que le premier interpréteur pour ASF+SDF fut réalisé, et c'est en 1993 qu'un premier compilateur (ASF2C) vit le jour. Le projet ASF+SDF s'est particulièrement intéressé aux mécanismes de définition de syntaxe d'un langage, à la génération automatique d'environnements de développement et aux techniques de *parsing* modulaires et incrémentales.

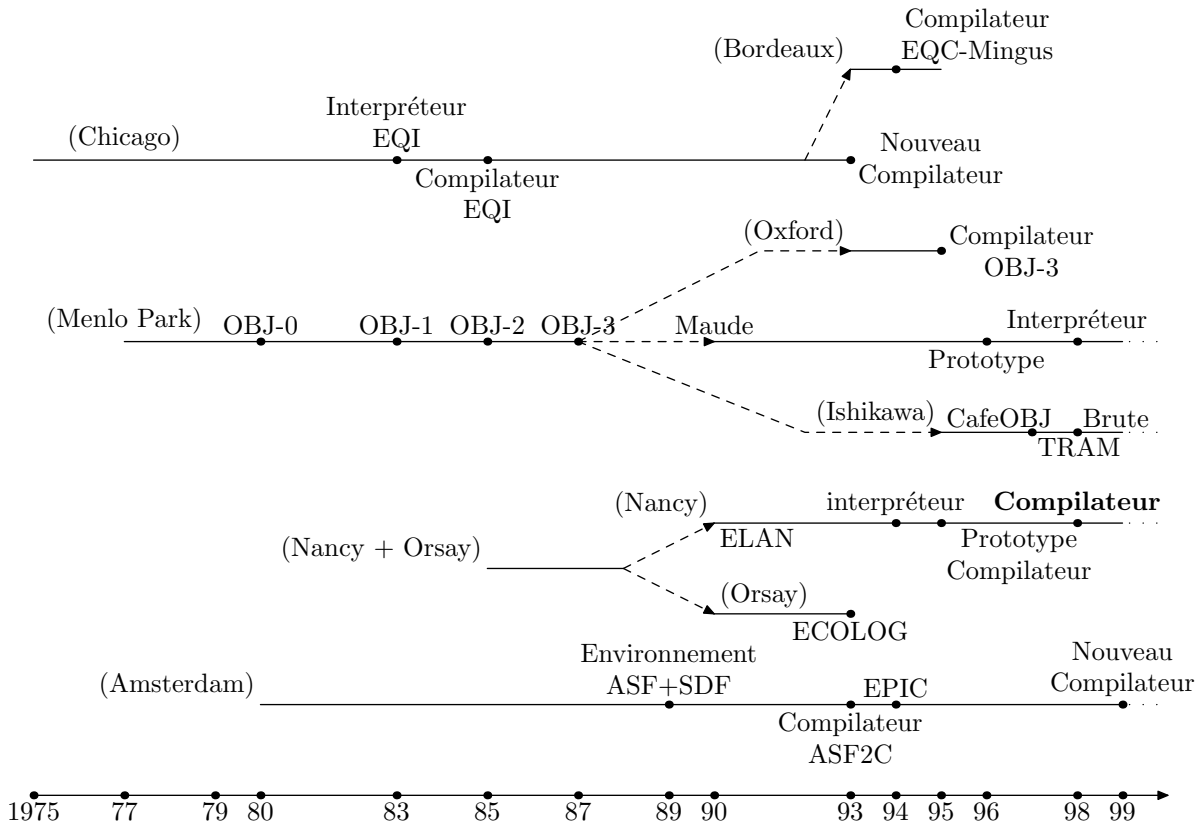


FIG. 1 – *Cette figure tente de retracer les développements logiciels « majeurs » effectués dans le domaine des langages de programmation fondés sur la logique de réécriture. Cette figure n'est évidemment pas exhaustive, mais permet de suivre l'évolution des principaux systèmes. EQI mis à part, il est intéressant de constater que la plupart des projets ont commencé tardivement l'étude des techniques de compilation, mais que l'attrait n'en est que plus intense.*

Réécriture et stratégies

Le principal intérêt des langages de programmation fondés sur la logique de réécriture est d'offrir des bases théoriques solides, une sémantique opérationnelle relativement simple[1] et une expressivité généralement puissante et agréable à utiliser. Ce dernier point est tout particulièrement intéressant lorsqu'on programme des algorithmes mathématiques complexes, ceci parce que les notations habituellement utilisées peuvent être réutilisées sans trop de changement. Cette absence de transcription, d'une notation à l'autre, diminue généralement le nombre d'erreurs et facilite la tâche du programmeur.

Considérons par exemple l'algorithme de complétion de Knuth-Bendix (1970), qui est souvent exprimé par les six règles de transformation suivantes :

$$
\begin{array}{lllll}
\textbf{Delete} & (E \cup \{s \simeq s\} \ ; \ R) & \mapsto\!\!\!\!\to & (E \ ; \ R) & \\
\textbf{Compose} & (E \ ; \ R \cup \{s \to t\}) & \mapsto\!\!\!\!\to & (E \ ; \ R \cup \{s \to u\}) & \\
 & & & \text{si } t \to_R u & \\
\textbf{Simplify} & (E \cup \{s \simeq t\} \ ; \ R) & \mapsto\!\!\!\!\to & (E \cup \{s \simeq u\} \ ; \ R) & \\
 & & & \text{si } t \to_R u & \\
\textbf{Orient} & (E \cup \{s \simeq t\} \ ; \ R) & \mapsto\!\!\!\!\to & (E \ ; \ R \cup \{s \to t\}) & \\
 & & & \text{si } s \succ t & \\
\textbf{Collapse} & (E \ ; \ R \cup \{s \to t\}) & \mapsto\!\!\!\!\to & (E \cup \{u \simeq t\} \ ; \ R) & \\
 & & & \text{si } s \xrightarrow[l \to r]{p} u \text{ avec } s \triangleright l & \\
\textbf{Deduce} & (E \ ; \ R) & \mapsto\!\!\!\!\to & (E \cup \{s \simeq t\} \ ; \ R) & \\
 & & & \text{si } s \simeq t \in cp(R) &
\end{array}
$$

Ces règles sont appliquées sur un couple (E,R) où E et R représentent respectivement des ensembles d'équations et de règles. L'ordre d'application des règles est important pour assurer une certaine équité et non-divergence du processus : la règle de déduction **Deduce** doit par exemple être appliquée seulement lorsqu'aucune autre alternative n'est possible. Cette *stratégie* d'application s'exprime habituellement par une expression régulière de la forme :

$$((\textbf{Collapse}^* \ ; \ \textbf{Compose}^* \ ; \ \textbf{Simplify}^* \ ; \ \textbf{Delete}^* \ ; \ \textbf{Orient}^*)^* \ ; \ \textbf{Deduce})^*$$

Dans le système ELAN, le codage des six règles précédentes se fait assez naturellement :

```
[Delete]    (E U {s=s} ; R)  => (E ; R )                          end
[Compose]   (E ; R U {s->t}) => (E ; R U {s->u})  if reduce(t->u)  end
[Simplify]  (E U {s=t} ; R)  => (E U {s=u} ; R)   if reduce(t->u)  end
[Orient]    (E U {s=t} ; R)  => (E ; R U {s->t})  if s > t         end
[Collapse]  (E ; R U {s->t}) => (E U {u=t} ; R)   if reduce(s->u)  end
[Deduce]    (E ; R)          => (E U {s=t} ; R)   if s=t in CP(R)  end
```

Une des originalités du langage ELAN est d'offrir la possibilité de spécifier en tant que telle la stratégie d'application des règles définies, ce qui permet de séparer clairement les règles de transformation et leur contrôle. Lorsqu'on ne dispose pas d'un tel langage de stratégie, l'ordre d'application des règles est souvent « codé » dans les règles de réécriture elles-mêmes, ce qui rend plus complexe et moins lisible le programme à écrire : les opérations de contrôle et de traitement

1. Ce qui permet de raisonner et de créer des outils de preuve automatique par exemple.

sont mélangées. En ELAN, on peut définir la stratégie donnée précédemment par la stratégie de réécriture suivante :

```
completion => repeat*(repeat*(repeat*(Collapse) ; repeat*(Compose) ;
              repeat*(Simplify) ; repeat*(Delete) ; repeat*(Orient)) ; Deduce)
```

Considérons maintenant la règle **Delete**, par exemple, qui exprime l'élimination des égalités triviales $\{s \simeq s\}$ de l'ensemble E. La simplicité d'expression d'une telle règle vient du fait que l'opérateur d'union $\cup$ est considéré associatif et commutatif. L'expression $E \cup \{s \simeq s\}$ prend en compte toutes les permutations possibles des éléments de E pour y rechercher l'égalité $\{s \simeq s\}$. Les langages Maude et ELAN permettent la définition de tels opérateurs, ce qui augmente considérablement leur expressivité et leur facilité à manipuler des structures d'ensembles ou de multi-ensembles par exemple. Le langage ASF+SDF propose quant à lui des opérateurs seulement associatifs, ce qui le rend plus apte à manipuler des structures de listes par exemple.

ELAN

Cette thèse s'inscrit dans le cadre de l'implantation du langage ELAN. Depuis la réalisation de l'interpréteur en 1993, le langage a été intensivement utilisé pour prototyper et réaliser de nombreuses applications telles que des langages de programmation avec contraintes, des résolveurs de contraintes et des outils de preuves de propriétés de programmes par exemple. Ce qui a plu dans un premier temps, c'est l'expressivité du langage, la possibilité de définir des notations infixées, des opérateurs associatifs et commutatifs, l'existence d'un préprocesseur permettant de générer automatiquement des systèmes de calcul, et surtout la puissance du langage de stratégie, qui permet d'exploiter entièrement l'aspect non-déterministe inhérent à la réécriture. Des applications majeures ont été réalisées, et rapidement le besoin d'un « moteur de réécriture » plus efficace s'est fait sentir. En 1995, Marian Vittek a commencé l'étude de techniques de compilation permettant d'améliorer les performances du langage, mais devant l'ampleur de la tâche et le temps qui lui était imparti, il n'a pu réaliser qu'un prototype de compilateur capable de traiter un sous-ensemble du langage ELAN. Les résultats étaient cependant très prometteurs, et surtout, il avait montré que des techniques de compilation particulières pouvaient rendre le système ELAN compétitif, en terme d'efficacité, avec bien d'autres langages de programmation.

Les difficultés pour compiler le langage ELAN sont principalement dues à la présence de stratégies non-déterministes : ces stratégies permettent d'explorer un sous-ensemble d'un espace de recherche en guidant finement l'application des règles. Lorsque cette exploration *échoue* en menant à une « impasse », une autre branche de l'espace de recherche doit être explorée.

Il existe cependant une deuxième source de difficulté : c'est la présence de symboles associatifs et commutatifs. Ces symboles nous amènent en effet à utiliser un algorithme de filtrage modulo les axiomes d'associativité et de commutativité. Le problème de filtrage est lui-même complexe, mais il introduit surtout un second niveau d'indéterminisme du fait qu'il peut exister plusieurs solutions à un problème donné. Une grande partie de la difficulté de la compilation du langage ELAN réside alors dans la mise en place d'un mécanisme capable de gérer efficacement et de façon cohérente ces deux sources de non-déterminisme.

Cette deuxième difficulté n'avait pas du tout été abordée par le prototype réalisé par Marian Vittek : les symboles associatifs et commutatifs ne pouvaient pas être compilés. Et pourtant, l'étude des spécifications écrites en ELAN a montré que l'utilisation de symboles associatifs et commutatifs a un impact réellement positif sur la qualité et la lisibilité des programmes, même si l'efficacité de ceux-ci est généralement inférieure à celle de programmes équivalents

n'utilisant pas de symboles associatifs et commutatifs. C'est pourquoi nous avons décidé d'étudier particulièrement comment compiler efficacement des spécifications ELAN utilisant des symboles associatifs et commutatifs.

Le réel défi de cette thèse est ainsi de proposer des techniques de compilation pour la totalité du langage ELAN, et de montrer qu'en pratique ces techniques permettent d'obtenir des programmes efficaces. L'intérêt de tels résultats est de montrer qu'un langage de spécification fondé sur la réécriture, ayant des bases théoriques solides et une grande expressivité, n'est pas condamné à rester à l'état de prototype et qu'il peut être utilisé pour réaliser des développements logiciels majeurs, tout en améliorant la qualité des logiciels ainsi construits.

Pour parvenir à notre objectif, nous étudions particulièrement les points délicats du processus de normalisation associatif-commutatif et nous proposons un nouvel algorithme, à base d'automates déterministes, pour compiler efficacement le filtrage syntaxique. Nous définissons ensuite différentes classes de règles pour lesquelles nous proposons un algorithme efficace de filtrage AC. L'étude du langage de stratégies nous conduit à implanter des primitives originales de gestion du *backtracking* et à définir un algorithme d'analyse du déterminisme permettant de réduire leur usage et d'améliorer encore les performances, tout en réduisant l'espace mémoire nécessaire. Enfin, l'implantation des méthodes proposées a donné lieu à l'élaboration de nombreuses optimisations théoriques et techniques qui peuvent être largement réutilisées pour implanter d'autres langages de programmation par réécriture. Cette thèse présente les algorithmes, l'architecture et le fonctionnement du compilateur, ainsi qu'une proposition d'environnement de spécification, fondée sur l'utilisation d'un format intermédiaire.

Présentation

Première partie : Environnement de spécification

Chapitre 1 : Langage de spécification ELAN. Cette thèse commence par une présentation intuitive du langage de spécification ELAN. Des exemples de programmes ELAN sont commentés pour inviter le lecteur à se familiariser avec les notions de grammaires, signatures, termes, règles et stratégies. En fin de chapitre, la notion de réécriture modulo les théories associatives et commutatives est introduite. C'est un des points qui sera particulièrement étudié dans la suite du document.

Chapitre 2 : Outils pour spécifier et programmer. Un langage de spécification ne devient un langage de programmation que si des outils informatiques existent pour le rendre exécutable sur une machine concrète. L'environnement de spécification ELAN est présenté et les notions de bibliothèque, parseur, interpréteur et compilateur sont introduites. Ce qui nous amène naturellement à comparer ELAN aux autres environnements de spécification liés à la réécriture.

Chapitre 3 : Plateforme de prototypage. L'environnement ELAN est un produit qui permet de mettre en pratique des idées ou des résultats issus de la recherche. Sa conception et son organisation doivent donc faciliter la mise en place rapide de nouvelles expériences. Une architecture de l'environnement, reposant sur l'existence d'un format intermédiaire d'échange, est ainsi proposée.

Deuxième partie : Compilation de la réécriture

Chapitre 4 : Méta-conception. La compilation est un art étudié dans de nombreux domaines, et c'est pourquoi elle est souvent perçue différemment d'un domaine à l'autre. Pour lever toute ambiguité, nous présentons ce que nous entendons par « interpréteur », « compilateur » et « machine abstraite », et les grandes lignes du compilateur que nous voulons définir sont présentées.

Chapitre 5 : Compilation du filtrage syntaxique. L'application de règles de réécriture est composée d'une étape de sélection, appelée *filtrage*. Les performances d'une procédure de normalisation, par réécriture, dépendent grandement du coût de l'algorithme de filtrage, et c'est pourquoi nous étudions particulièrement comment compiler un tel algorithme.

Chapitre 6 : Compilation du filtrage associatif-commutatif. Les algorithmes de filtrage associatif-commutatif ont été largement étudiés dans le passé, aussi bien pour réaliser des outils de déduction automatique que des outils de calcul intensif. Ici nous proposons de spécialiser la conception d'un tel algorithme dans le cadre d'une procédure de normalisation par réécriture. Ce contexte particulier nous amène à définir une nouvelle structure de données compacte, qui permet de réduire le coût des algorithmes impliquées dans la procédure de filtrage associatif-commutatif.

Chapitre 7 : Gestion du non-déterminisme. La nature du langage ELAN fait de sa compilation un réel défi. La présence de stratégies et d'opérateurs associatifs-commutatifs, source de double non-déterminisme, nous amène à étudier des schémas de compilation originaux reposant sur la définition de primitives de gestion du non-déterminisme. Ce chapitre technique présente deux nouvelles fonctions, setChoicePoint et fail, qui permettent d'intégrer, de manière transparente, la gestion de points de choix au langage C.

Chapitre 8 : Compilation des règles et des stratégies. Des schémas de compilation, intégrant la sélection d'une règle de réécriture, l'évaluation des conditions, l'application de stratégies et la constuction du terme réduit sont présentés. L'intérêt est de présenter un mécanisme uniforme de gestion du non-déterminisme lié aux stratégies et à la présence d'opérateurs associatifs-commutatifs.

Chapitre 9 : Analyse du déterminisme. La présence de non-déterminisme est souvent source d'inefficacité. Dans ce chapitre, nous présentons un algorithme permettant d'inférer un mode de déterminisme particulier pour chaque règle ou stratégie. Le mode inféré est ensuite utilisé pour modifier et améliorer les schémas de compilation présentés dans le chapitre 8. L'efficacité du code généré se voit ainsi améliorée, et sa consommation mémoire réduite.

Troisième partie : Implantation d'un compilateur

Chapitre 10 : Architecture logicielle. Les idées présentées dans cette thèse sont mises en pratique à travers la réalisation d'un compilateur. Les problèmes liés à la compilation modulaire de systèmes de réécriture sont présentés. Les solutions retenues, l'organisation générale du compilateur ainsi que son fonctionnement sont aussi présentés.

Chapitre 11 : Support d'exécution. La réalisation d'un compilateur consiste essentiellement à étudier des schémas de génération de programmes. Mais l'étude de l'environnement d'exécution des programmes générés est aussi importante. Ce chapitre aborde les problèmes liés à la représentation des données, la définition d'opérateurs prédéfinis par le langage de spécification ainsi que différentes techniques de gestion mémoire.

Chapitre 12 : Expériences pratiques. Ce chapitre montre l'intérêt des méthodes imaginées en évaluant la « qualité » des programmes engendrés par le compilateur. Nous présentons ainsi des spécifications ELAN écrites dans différents styles de programmation et nous étudions particulièrement, après compilation, la consommation mémoire, le « degré de compilation », les performances et l'apport des techniques de compilation imaginées.

Première partie

Environnement de spécification

Chapitre 1

Langage de spécification ELAN

1.1 Grammaire et signature 11
1.2 Termes et règles de réécriture conditionnelles 13
1.3 Stratégies d'application 15
1.4 Règles et stratégies 16
1.5 Opérateurs Associatifs et Commutatifs 21
1.6 Modularité 23

Un des champs couverts par le domaine des spécifications algébriques est celui des descriptions formelles de types de données abstraits. Une spécification algébrique est généralement composée de plusieurs parties : les signatures qui décrivent la structure des types de données utilisés, les sortes des données, les opérations applicables sur ces données et les expressions, ou formules logiques, qui définissent les propriétés des opérations. Un formalisme de spécification algébrique est caractérisé par la syntaxe des signatures, les formules logiques autorisées et par les opérations additionnelles qui permettent de développer des spécifications.

ELAN est un formalisme de spécification modulaire du premier ordre qui comporte des signatures multi-sortées, des règles de réécritures conditionnelles et des stratégies. Une des particularités d'ELAN est de permettre l'utilisation de grammaires hors contexte pour décrire les signatures. Ceci permet de définir et d'utiliser des opérateurs infixés qui font d'ELAN un langage agréable à utiliser pour spécifier des structures de données complexes telles que celles utilisées dans des prouveurs automatiques, des résolveurs de contraintes ou des outils de transformation de programmes par exemple.

Ce chapitre n'a pas pour ambition de présenter en détail toutes les constructions autorisées par le langage ELAN, et encore moins de servir de mode d'emploi du système logiciel existant. Au contraire, certaines constructions sont volontairement occultées afin de ne pas perdre le lecteur et de présenter, de manière simple et intuitive, les caractéristiques principales du langage qui seront nécessaires pour avoir une bonne compréhension des solutions proposées dans les chapitres suivants. Le lecteur est invité à se référer à la thèse de Marian Vittek (1994) ou au manuel ELAN (Borovanský, Kirchner, Kirchner, Moreau et Vittek 1997) pour avoir plus de détails concernant les fondements et l'utilisation du système ELAN.

1.1 Grammaire et signature

La première partie d'une spécification ELAN consiste en la définition des sortes utilisées, la liste des modules importés et un ensemble de règles de grammaires hors contexte pour déclarer

les symboles de fonctions.

Considérons, par exemple, la signature d'un module définissant les booléens : appelons `Bool` la sorte des objets manipulés, `vrai` et `faux` les valeurs de vérité et `et`, `ou` et `non` les opérations définies sur l'algèbre des booléens. La déclaration d'une telle signature s'écrit de la manière suivante en ELAN :

```
module booleen
sort Bool; end
operators global
  vrai   :             Bool;
  faux   :             Bool;
  @ et @ : (Bool Bool) Bool;
  @ ou @ : (Bool Bool) Bool;
  non @  : (Bool )     Bool;
end
```

Le caractère `@` est un symbole spécial qui indique la place d'un argument dans la définition d'un opérateur. Les sortes des arguments sont données par une liste de sortes entre parenthèses.

Un des problèmes posés par cette définition est que certains termes peuvent être ambigus. Le terme `vrai et vrai et vrai`, par exemple, est un terme de sorte `Bool` qui peut se représenter de deux manières différentes suivant que l'opérateur `et` est associatif gauche ou droite. Les attributs `assocLeft` et `assocRight` peuvent être utilisés pour déclarer un opérateur associatif gauche ou associatif droit et lever ainsi ce type d'ambiguité. L'utilisation de l'attribut `pri` permet de donner des priorités aux opérateurs et ainsi exprimer le fait qu'un opérateur soit prioritaire par rapport à un autre. Il existe aussi un mécanisme d'alias permettant de définir un même opérateur de plusieurs manières différentes.

Afin d'éliminer les ambiguïtés, la grammaire précédente peut se réécrire de la manière suivante :

```
module booleen
sort Bool; end
operators global
  vrai     :             Bool;
  faux     :             Bool;
  @ et @   : (Bool Bool) Bool assocLeft pri 100;
  @ ou @   : (Bool Bool) Bool assocLeft pri 100;
  (@ et @) : (Bool Bool) Bool assocLeft pri 100 alias @ et @:;
  (@ ou @) : (Bool Bool) Bool assocLeft pri 100 alias @ ou @:;
  non @    : (Bool )     Bool           pri 200;
  non (@)  : (Bool )     Bool           pri 200 alias non @:;
end
```

Il est parfois nécessaire de définir une injection d'une sorte vers une autre. Ceci peut s'exprimer facilement en utilisant un opérateur « sans nom ». Supposons que nous voulions définir une sorte `Contrainte` permettant de représenter des formules logiques. Certaines de ces formules peuvent se simplifier en des contraintes élémentaires correspondant aux valeurs de vérité `vrai` et `faux`. Il est alors pratique de dire que toute expression de sorte `Bool` est aussi une expression de sorte `Contrainte`. Cela s'exprime en ELAN de la façon suivante :

```
module contrainte
sort Contrainte; end
operators global
  @ : (Bool) Contrainte;
end
```

On dit alors que la sorte `Bool` est « injectée » dans la sorte `Contrainte`.

D'un point de vue plus formel, une signature Σ est un couple $(\mathcal{S},\mathcal{F})$ où $\mathcal{S}$ est un ensemble de sortes et $\mathcal{F}$ est un ensemble de symboles de fonctions sur lequel sont définies les applications $\mathcal{D}om : \mathcal{F} \mapsto \mathcal{S}$, $\mathcal{C}od : \mathcal{F} \mapsto \mathcal{S}$ et $\# : \mathcal{F} \mapsto \mathbb{N}$ retournant respectivement le domaine, le codomaine et l'arité d'un symbole de fonction.

Soient $f \in \mathcal{F}$, $s \in \mathcal{S}$ et $(s_1, \ldots, s_n) \in \mathcal{S}^n$ tels que $\mathcal{D}om(f) = (s_1, \ldots, s_n)$, $\mathcal{C}od(f) = s$ et $\#f = n$, on dit que le symbole f a pour profil : $(s_1, \ldots, s_n) \mapsto s$.

1.2 Termes et règles de réécriture conditionnelles

Les symboles définis dans la signature peuvent être utilisés pour construire des termes.

Étant données une signature $\Sigma = (\mathcal{S},\mathcal{F})$ et $\mathcal{X}$ une famille d'ensembles de variables $\mathcal{X}_s$ de sorte $s \in \mathcal{S}$, l'ensemble des termes $\mathcal{T}_s(\mathcal{F},\mathcal{X})$ de sorte s est le plus petit ensemble contenant $\mathcal{X}_s$ et tel que $f(t_1, \ldots, t_n)$ est dans $\mathcal{T}(\mathcal{F},\mathcal{X}) = \bigcup_{s\in\mathcal{S}} \mathcal{T}_s(\mathcal{F},\mathcal{X})$ pour toute fonction f de profil $(s_1, \ldots, s_n) \mapsto s$ et $t_i \in \mathcal{T}_{s_i}(\mathcal{F},\mathcal{X})$, pour $i \in [1..n]$.

Je suppose connue la notion de *position* dans un terme (la position vide, qui correspond à la racine est notée ϵ). Le sous-terme de t à la position ω est noté $t_{|\omega}$. Le remplacement dans t, de $t_{|\omega}$ par t' est noté $t[t']_\omega$.

Un terme est dit *clos* s'il ne contient pas de variable et l'ensemble des termes clos se note $\mathcal{T}(\mathcal{F})$.

Nous avons vu comment les grammaires et les signatures permettent de définir et de construire la structure algébrique des données, mais nous n'avons rien dit concernant le *sens* des opérations ainsi définies. En ELAN, le mécanisme d'évaluation élémentaire repose sur la *réécriture* : les règles de réécriture sont des paires de termes (l,r) notées $l \to r$ ou `l => r` et sont utilisées pour définir une relation entre deux termes clos. Nous pouvons ainsi définir un ensemble de six règles de réécriture qui permettent de simplifier en `vrai` ou `faux` n'importe quelle expression booléenne du module `booleen` :

```
rules for Bool
  P : Bool;
global
  [] vrai ou P => vrai end
  [] faux ou P => P    end
  [] vrai et P => P    end
  [] faux et P => faux end
  [] non vrai  => faux end
  [] non faux  => vrai end
end
```

Un tel ensemble de règles est appelé *système de réécriture* et sert à simplifier des termes clos construits sur la signature de ce système. Les règles, elles, ne sont pas forcément composées de termes clos. Dans l'exemple précédent, certains membres gauches de règles contiennent une

variable `P`. Pour pouvoir appliquer une règle sur un terme clos, appelé *sujet*, il faut que l'on puisse remplacer les variables de son membre gauche par des termes clos, de telle sorte que ce nouveau membre gauche soit « égal » au *sujet*. On dit alors que le membre gauche de la règle *filtre* vers le sujet qui devient un *radical*. L'assignement qui remplace chaque variable par un terme clos est appelé *substitution* ou *filtre*.

Plus formellement, une substitution σ sur $\mathcal{T}(\mathcal{F},\mathcal{X})$ est un endomorphisme de $\mathcal{T}(\mathcal{F},\mathcal{X})$ qui s'écrit $\sigma = (x_1 \mapsto t_1 \circ \cdots \circ x_n \mapsto t_n)$ lorsque les images de x_i pour $i = 1, \ldots, n$ sont des $t_i \neq x_i$. Une des propriétés fondamentales des substitutions est que pour tous termes $t_1, \ldots, t_n \in \mathcal{T}(\mathcal{F},\mathcal{X})$ et pour tout symbole $f \in \mathcal{F}$:

$$f(t_1, \ldots, t_n)\sigma = f(t_1\sigma, \ldots, t_n\sigma) \quad \text{(c'est la propriété d'endomorphisme)}$$

Une substitution σ appliquée au terme t est notée $t\sigma$ ou $\sigma(t)$.

Lorsqu'il existe une règle dont le membre gauche filtre vers le sujet, celle-ci peut s'appliquer et réduire le sujet. Le mécanisme d'application d'une règle consiste simplement à remplacer le sujet par le membre droit de la règle sur lequel est appliqué le filtre.

Lorsqu'aucune règle n'est applicable sur un terme, on dit qu'il n'est plus réductible et qu'il est en *forme normale*. Le système précédent sur les expressions booléennes est intéressant parce qu'on peut montrer qu'il a les propriétés suivantes :

- peu importe le terme de départ, on sait qu'une de ses formes normales sera obtenue après un nombre fini d'étapes de réécritures. On dit alors que le système *termine* ;
- pour un terme donné, l'ordre d'application des règles de réécriture et la position où s'applique une règle n'ont aucune influence sur le résultat : on obtient toujours la même forme normale. Le système est alors dit *confluent*.

D'un point de vue théorique, il est intéressant de considérer des systèmes terminants et confluents parce qu'on sait alors que les spécifications proposées permettent de calculer des résultats en un temps fini et ceci quels que soient les termes d'entrée. On sait de plus que les résultats retournés seront toujours les mêmes et ceci quelle que soit la façon d'implanter les spécifications. Malheureusement, la pratique montre qu'il est assez difficile d'écrire des spécifications à base de règles de réécriture qui soient confluentes et terminantes. D'une manière générale, deux grandes voies ont été étudiées pour aider les informaticiens à écrire des spécifications confluentes et terminantes :

- la première consiste à créer des outils permettant d'aider le programmeur à vérifier qu'une spécification donnée est confluente et terminante (Knuth et Bendix 1970, Kirchner et Moreau 1995). La réalisation de tels outils reste cependant complexe dans la mesure où le problème est indécidable. De plus, les résultats trouvés à ce jour montrent qu'il est difficile d'appliquer ces outils aux spécifications de grande taille : d'une manière générale, les propriétés de terminaison ou de confluence ne sont pas modulaires, ce qui signifie qu'étant donnés deux systèmes de réécritures terminants et confluents, leur union n'a pas forcément les mêmes propriétés. On imagine alors facilement les difficultés rencontrées pour montrer qu'une spécification composée de plusieurs centaines de modules termine bien.
- la deuxième voie, qui n'est pas antagoniste avec la première, consiste à aborder le problème par l'autre bout : puisqu'il est difficile de vérifier qu'une spécification donnée est bien terminante et confluente, l'approche consiste à étudier les langages de spécifications eux-mêmes pour améliorer leur expressivité, leur sûreté et permettre plus facilement aux programmeurs d'écrire des spécifications correctes, confluentes et terminantes. Le langage ELAN fait partie de ces langages. Il est bien sûr possible d'écrire des spécifications incor-

rectes avec des langages de « haut niveau » mais cela arrive moins souvent qu'en utilisant l'assembleur par exemple.

Afin d'illustrer les difficultés et les solutions proposées pour écrire des spécifications terminantes, essayons par exemple de définir la fonction *factorielle* en ELAN :

```
rules for int
  n : int;
global
  [] fact(0) => 1           end
  [] fact(1) => 1           end
  [] fact(n) => n*fact(n-1) end
end
```

Pour définir la fonction factorielle, nous avons importé le module `int` qui définit la sorte du même nom permettant de représenter des entiers. Des opérations élémentaires telles que l'addition, la soustraction et la multiplication sont pré-définies. Le système précédent permet de calculer des valeurs de la fonction factorielle, mais le résultat n'est pas entièrement satisfaisant. Calculons la valeur de la fonction factorielle en 1 : il suffit de calculer la forme normale du terme `fact(1)`. Le problème, ici, est que le calcul de la forme normale peut ne pas terminer : `fact(1)` peut se simplifier en `1` en appliquant la deuxième règle, mais si on applique la troisième règle, cela peut nous amener à calculer `fact(0)`, puis `fact(-1)`, `fact(-2)`, *etc.*

Pour aider le programmeur à écrire des systèmes de réécriture terminants et confluents, des *conditions* peuvent être ajoutées pour contrôler l'application des règles. On parle alors de règles de réécriture *conditionnelles*. Il suffit d'ajouter une condition (introduite par le mot clé `if`) au système précédent pour le rendre terminant et confluent :

```
rules for int
  n : int;
global
  [] fact(0) => 1                  end
  [] fact(1) => 1                  end
  [] fact(n) => n*fact(n-1) if n>1 end
end
```

Dans ce cas, la troisième règle ne peut plus s'appliquer pour réduire le terme `fact(1)` parce que la condition `if n>1` n'est plus satisfaite. On peut d'ailleurs montrer que ce dernier système est bien terminant et confluent. Mais les techniques pour montrer la confluence et la terminaison des systèmes de réécriture conditionnelle sont encore plus complexes.

1.3 Stratégies d'application

L'étude de la réécriture et le développement d'ELAN s'intègrent dans le cadre du génie logiciel, en essayant d'améliorer la qualité des environnements de développement et des logiciels ainsi produits. Mais l'aspect non-déterministe de la réécriture (les règles de réécriture peuvent s'appliquer dans n'importe quel ordre et à n'importe quelle position du terme à réduire) n'est pas vraiment compatible avec la volonté de réaliser des logiciels sûrs. En effet, même s'il existe des algorithmes permettant de prouver que certains systèmes sont confluents et terminants, d'une manière générale, ces problèmes ne sont pas décidables parce que isomorphes à l'indécidabilité de l'arrêt des machines de Turing (Turing 1936, Delahaye 1995) : il existe toujours des systèmes

de réécriture dont on ne peut prouver ni la terminaison, ni la non-terminaison.

C'est principalement ce qui a amené les théoriciens à introduire la notion de *stratégie* d'application pour mieux contrôler l'application des règles de réécriture. Les stratégies les plus connues sont les suivantes :

- la stratégie de *parcours intérieur gauche* (*leftmost-innermost*) sélectionne le radical le plus à gauche et le plus interne à chaque étape de réécriture ;
- la stratégie de *parcours intérieur parallèle* (*parallel-innermost*) sélectionne tous les radicaux les plus internes ;
- la stratégie de *parcours extérieur gauche* (*leftmost-outermost*) sélectionne le radical le plus à gauche et le plus externe à chaque étape de réécriture ;
- la stratégie de *parcours extérieur parallèle* (*parallel-outermost*) sélectionne tous les radicaux les plus externes.

Dans l'environnement ELAN, c'est la stratégie *leftmost-innermost* qui a été retenue comme stratégie de normalisation. Il existe cependant un autre moyen de contrôler l'application des règles dans ELAN, celui-ci consiste à utiliser des stratégies « définies par l'utilisateur ».

Un lecteur attentif aura sans doute remarqué que les règles de réécriture définies précédemment commencent toutes par un crochet ouvrant et un crochet fermant `[]`. Il s'agit en fait d'un emplacement permettant de *nommer* une règle particulière. Lorsque cet emplacement est laissé vide, comme c'était le cas jusqu'à présent, on parle alors de règles *non nommées*. Un système de calcul ELAN est composé de trois parties :

- des règles non nommées qui sont appliquées le plus souvent possible en suivant la stratégie leftmost-innermost. La position où s'applique une règle est déterminée par la stratégie, par contre, le choix de la règle à appliquer n'est pas défini ;
- des règles nommées qui ne sont appliquées que lorsque le programmeur le demande explicitement. Ces règles sont toujours appliquées à la racine des termes, mais cette fois-ci, le choix de la règle à appliquer peut être contrôlé par l'utilisateur ;
- des stratégies qui sont des expressions construites à partir d'opérateurs élémentaires. Les stratégies utilisent les noms (appelés aussi *labels* ou *étiquettes*) donnés aux règles pour ordonnancer et contrôler leur application.

1.4 Règles et stratégies

Une des originalités d'ELAN est de permettre à l'utilisateur de contrôler l'application des règles de réécriture en définissant des stratégies. À partir des noms de règles, il est ainsi possible de construire des stratégies qui retournent un ou plusieurs résultats, d'ordonnancer l'application des règles et de répéter aussi longtemps que possible l'application d'une règle ou d'une stratégie.

Une règle nommée est ainsi considérée comme une stratégie élémentaire et le résultat de l'application d'une règle nommée lab sur un terme t retourne l'ensemble des termes atteignables en appliquant la règle lab. Si aucune règle étiquetée par lab ne peut s'appliquer, on dit alors que la stratégie *échoue*. Pour comprendre comment l'application d'une seule règle à la racine d'un terme peut retourner plusieurs résultats il faut savoir qu'un mécanisme d'*évaluation locale* existe. Sa description et son utilisation seront détaillées un peu plus loin dans ce chapitre. Dans un premier temps, nous pouvons considérer que c'est une construction qui permet de déclencher l'application d'une stratégie. Si celle-ci retourne plusieurs résultats, la règle nommée considérée retourne elle aussi plusieurs résultats.

Nous venons de voir que toute règle nommée est une stratégie, c'est pourquoi, dans la suite de la présentation du langage de stratégies, nous ne considérons que des opérateurs qui ont des stratégies en argument pour construire de nouvelles stratégies :

- l'opérateur de concaténation, noté `;`, permet de composer l'application de deux stratégies S_1 et S_2. La stratégie $S_1; S_2$ échoue si S_1 échoue, sinon elle retourne tous les résultats de la stratégie S_2 appliquée aux résultats de S_1. La stratégie échoue également si S_2 échoue pour chaque résultat de S_1 ;
- l'opérateur dk est une abréviation de *dont know choose*. Il est particulier dans la mesure où son arité est variable : dk$(S_1, \ldots, S_n)$ sélectionne toutes les stratégies données en argument et retourne, pour chacune d'elles, l'ensemble des résultats possibles. Si toutes les stratégies $S_1, \ldots, S_n$ échouent, la stratégie dk$(S_1, \ldots, S_n)$ échoue elle aussi ;
- l'opérateur dc tient son nom de *dont care choose*. À la différence de dk, il ne sélectionne, parmi sa liste d'arguments, qu'une seule stratégie S_i qui n'échoue pas. Il retourne ensuite l'ensemble des résultats provenant de l'application de S_i. La méthode de sélection de la stratégie S_i n'est pas spécifiée et peut être considérée comme non-déterministe ;
- lorsque l'ordre de sélection a une importance particulière, on peut alors utiliser l'opérateur first qui sélectionne la première stratégie qui n'échoue pas en essayant les stratégies de la gauche vers la droite : lorsque first$(S_1, \ldots, S_n)$ sélectionne la stratégie S_i, c'est que toutes les stratégies $S_1, \ldots, S_{i-1}$ ont échoué et l'ensemble des résultats de l'application de S_i est alors retourné ;
- il arrive qu'on ne soit intéressé que par un seul résultat, dans ce cas il est possible d'utiliser les opérateurs first one et dc one qui sélectionnent (avec ou sans ordre) une stratégie qui n'échoue pas et retournent au plus un résultat. Celui-ci est choisi de manière non-déterministe parmi l'ensemble des résultats possibles ;
- la stratégie id est la stratégie qui ne fait rien, mais qui n'échoue jamais ;
- à l'inverse, la stratégie fail échoue tout le temps et ne retourne jamais de résultats ;
- la stratégie repeat*(S) applique répétitivement la stratégie S jusqu'à ce qu'elle échoue et retourne le dernier résultat obtenu. Cette stratégie est particulière dans la mesure où elle n'échoue jamais : zéro application de S est possible, et dans ce cas, le terme initial est retourné ;
- la stratégie iterate*(S) est similaire à repeat*(S) mais retourne les résultats intermédiaires des applications successives de S.

Définissons un module permettant de construire des listes. Dans ce module, la liste vide est notée `nil` et l'opérateur infixe de concaténation est noté `.` :

```
module liste
sort Element Liste; end
operators global
  a          :                  Element;
  b          :                  Element;
  c          :                  Element;
  nil        :                  Liste;
  @.@        : (Element Liste) Liste;
end
```

Le terme `a.b.nil` permet ainsi de représenter la liste contenant les éléments `a` et `b`. Pour se familiariser un peu plus avec les constructions du langage, essayons d'écrire un programme

capable d'extraire tous les éléments d'une liste. Habituellement, il suffit d'écrire une fonction qui extrait l'élément de tête et qui s'applique récursivement sur le reste de la liste. Il est bien sûr possible de suivre la même approche en ELAN, mais ce ne serait pas tellement dans « l'esprit » du langage. En effet, une des originalités d'ELAN est de permettre une séparation claire entre les fonctions qui manipulent les données (appelées règles de réécriture) et les fonctions qui contrôlent l'application de ces fonctions (appelées stratégies). Nous pouvons ainsi définir des règles qui permettent d'extraire la tête et la queue d'une liste et définir une stratégie qui décrit comment appliquer ces règles afin d'obtenir l'*ensemble* des éléments qui composent la liste. Dans la phrase précédente, est-ce un hasard que le mot *ensemble* soit en italique ? En fait non, c'est parce que nous sommes en mesure d'extraire tous les éléments d'une liste, mais nous ne savons pas encore comment représenter cet ensemble de résultats. Nous pourrions mémoriser les éléments dans une nouvelle liste mais cela ne nous avancerait pas beaucoup. Supposons que nous voulions appliquer un *traitement* à chaque élément qui compose la liste, faut-il combiner la fonction d'extraction avec le traitement ? Ici, la notion d'ensemble de résultats n'a pas besoin d'être explicitement représentée, elle fait partie intégrante du mécanisme d'évaluation des stratégies. Comme nous l'avons vu précédemment, d'un point de vue théorique, une stratégie retourne un *ensemble* de résultats. Mais d'un point de vue pratique, les résultats sont retournés « à la demande », ce qui signifie qu'une stratégie commence par retourner un seul résultat (si elle n'échoue pas) et si plus tard un échec se produit, un mécanisme de *retour arrière* (appelé aussi *gestion du non-déterminisme* ou *backtracking*) se met en place et provoque l'extraction des solutions qui n'ont pas encore été retournées. On peut ainsi considérer que la notion d'*ensemble* est une structure interne au système.

Voyons comment traiter notre exemple en ELAN : commençons par définir deux règles nommées qui retournent respectivement l'élément en tête de liste et la liste qu'il reste à inspecter. On pourrait définir les deux règles suivantes :

```
[extractrule1] extract(element.liste) => element        end
[extractrule2] extract(element.liste) => liste          end
```

mais dans ce cas, les sortes des membres droits ne seraient pas identiques. Afin de rendre homogène les sortes impliquées dans ces deux règles, la signature de notre système doit être étendue par l'ajout de l'opérateur `extract(@) : (Liste) Element`.

Ce constructeur permet de considérer l'objet `extract(a.b.nil)` comme étant un terme de sorte `Element`. Les deux règles nommées se définissent alors de la manière suivante :

```
[extractrule1] extract(element.liste) => element        end
[extractrule2] extract(element.liste) => extract(liste) end
```

Étant donnée une liste (de sorte `Element`), l'application de la règle `extractrule2` retourne la même liste privée de l'élément de tête. Les applications successives de la deuxième règle retournent ainsi autant de listes qu'il y a d'éléments dans la liste initiale (c'est vrai si on considère que la liste initiale correspond à 0 application d'`extractrule2`). Il suffit alors d'appliquer la règle `extractrule1` sur chaque sous-liste obtenue pour en extraire l'élément de tête. C'est précisément cette idée qui est exprimée par la stratégie suivante :

```
[] listExtract => iterate*(dc one(extractrule2)) ; dc one(extractrule1) end
```

À titre d'exemple, étudions l'application de la stratégie `listExtract` sur le terme `extract(a. b.nil)`. Dans un premier temps, la règle `extractrule2` est appliquée 0 fois (c'est le premier résultat d'`iterate*`), ce qui ne modifie pas le terme courant, puis la première règle

est appliquée pour retourner le premier résultat de la stratégie : l'élément `a`. Lorsqu'une autre solution est demandée, l'itération continue et la règle `extractrule2` est appliquée sur le terme résultat de sa dernière application, à savoir : `extract(a.b.nil)`. Le terme se réduit en `extract(b.nil)` puis l'élément `b` est retourné. Si une autre solution est de nouveau demandée, l'application d'`extractrule2` réécrit le terme `extract(b.nil)` en `extract(nil)`, mais cette fois ci, la première règle ne peut pas s'appliquer, il y a donc un *échec* dans la deuxième partie de la stratégie. Cet échec provoque la demande d'une autre solution, mais l'itération est terminée, c'est pourquoi la stratégie toute entière échoue : tous les éléments de la liste ont été extraits.

Afin d'intégrer la gestion des stratégies et de permettre l'exploitation des résultats, la syntaxe et la sémantique des règles de réécriture à été étendue. La structure d'une règle ELAN est la suivante :

< *règle* > `::=`
`"[" [` <*étiquette*> `] "]"` <*terme*> `"=>"` <*terme*> `{` <*évaluation locale*> `}`*

< *évaluation locale* > `::=`
if <*terme booléen*>
| **where** <*nom de variable*> `":="` `"("` `[` <*stratégie*> `] ")"` <*terme*>
| **where** `"("` <*sorte*> `")"` <*terme*> `":="` `"("` `[` <*stratégie*> `] ")"` <*terme*>
| **choose**
`{` **try** `{` <*évaluation locale*> `}`$^+$ `}`$^+$
end

Il faut noter qu'une règle se décompose en quatre composantes principales :

- une éventuelle *étiquette* qui permet de donner un nom à la règle pour en faire une stratégie élémentaire ;
- un membre gauche utilisé dans l'étape de filtrage pour savoir si la règle peut s'appliquer ou non ;
- un membre droit qui décrit la structure du terme réduit ;
- une liste d'évaluations locales qui permettent de déclencher des stratégies, de mettre en facteur des suites de calculs ou de spécifier des conditions d'applications de la règle.

La simplification d'un terme clos commence alors par une étape de filtrage permettant d'éliminer les règles ne pouvant pas s'appliquer sur le terme. Une règle est ensuite sélectionnée parmi les candidates restantes, ce qui permet de calculer la substitution associée au problème de filtrage considéré. Les évaluations locales sont alors évaluées les unes à la suite des autres (de haut en bas) jusqu'à atteindre la dernière ; c'est seulement à ce moment là que la règle peut s'appliquer et que le membre droit est construit. Il existe actuellement trois types d'évaluations locales qui permettent d'augmenter de manière significative l'expressivité des systèmes de réécriture. Une condition est une expression booléenne `c` introduite par le mot clé `if`. De son évaluation dépend l'application de la règle courante : le terme `c` est mis en forme normale puis comparé à la valeur de vérité `true` pré-définie par le système. En cas d'égalité, on dit que la condition est satisfaisable et le calcul des évaluations locales se poursuit. En cas d'inégalité, on dit que l'évaluation locale échoue, ce qui déclenche le mécanisme de *retour arrière* (*backtracking*) : les évaluations locales précédentes sont réévaluées pour en extraire d'autres solutions. Cela revient à « changer de branche » au cours d'une exploration d'un arbre de recherche. Si aucune autre solution n'est trouvée, on dit que l'application de la règle courante échoue et une autre règle est sélectionnée.

La construction `where v:=(S) t` (*affectation locale*) permet de déclencher l'application d'une stratégie. Dans un premier temps, le terme `t` est mis en forme normale en n'utilisant que des

règles non nommées, la stratégie `S` est ensuite appliquée sur le terme en forme normale. D'un point de vue pratique, seul un résultat de l'application de `S` est calculé et affecté à la variable `v`. Si la stratégie échoue, l'évaluation locale échoue également et le mécanisme de retour arrière se met en place. Lorsqu'à la suite d'un échec une affectation locale redevient « active », la forme normale du terme `t` n'a pas besoin d'être recalculée parce qu'à la suite de la première évaluation, ce résultat intermédiaire est mémorisé par le mécanisme de gestion des retours arrières. La réactivation d'une affectation locale consiste alors à poursuivre l'évaluation de la stratégie `S` pour en extraire une nouvelle solution, si elle existe.

Il existe une extension appelée *condition de filtrage* (*matching condition*) qui permet de remplacer la variable `v` par un terme `p` quelconque. Le mécanisme d'évaluation est sensiblement le même que précédemment, mis à part le fait que le résultat de l'application de la stratégie `S` n'est plus simplement affecté à la variable `v` mais filtré par le motif `p`. Les variables du terme `p` sont alors instanciées par leurs valeurs résultant du filtrage. Lorsque la règles suivante est appliquée sur le terme `a.b.c.nil`, les variables `premier`, `second` et `reste` sont respectivement instanciées par les termes `a`, `b` et `c.nil`. La condition des filtrage est une construction expressive permettant de décomposer un terme pour accéder facilement à ses sous-termes.

```
  [] liste => premier.second.nil
       where (Liste) premier.second.reste :=() liste
```

Le troisième type d'évaluation locale est de loin le plus complexe : il permet de mettre en facteur des séquences de calcul en évitant d'avoir à écrire plusieurs règles de réécriture pour décrire un algorithme. La construction `choose try ... end` offre la possibilité de créer des sous-listes d'évaluations locales précédées par le mot clé `try`. On peut ainsi décrire le calcul de la fonction factorielle en n'écrivant qu'une seule règle :

```
rules for int
  n      : int;
  result : int;
global
  [] fact(n) => result
       choose
         try if n==0 or n==1
             where result:=() 1
         try if n>1
             where result:=() n*fact(n-1)
       end
  end
end
```

Pour cet exemple, la transformation proposée n'a aucun intérêt parce que les problèmes de filtrage `fact(0) => ...` et `fact(1) => ...` sont remplacés par la condition `if n==0 or n==1`. Mais plaçons nous dans un cadre plus complexe et imaginons que la description d'un algorithme nécessite plusieurs règles ayant le même membre gauche (cela arrive fréquemment en pratique) :

```
rules for int
  x,y,z : ...
global
  [] f(x) => r1(z)
       where y:=() g(x)
       where z:=(s1) x
  end
  [] f(x) => r2(z)
       where y:=() g(x)
       where z:=(s2) x
  end
end
```

Pour réduire le terme `f(a)` par exemple, une règle est sélectionnée. Imaginons que ce soit la première et supposons que l'application de la stratégie `s1` sur le terme `a` échoue. Dans ce cas, la deuxième règle est essayée et la forme normale du terme `g(a)` doit une nouvelle fois être calculée. Pour éviter ce calcul redondant, on peut transformer le système de la manière suivante :

```
rules for int
  x,y,z,result : ...
global
  [] f(x) => result
       where y:=() g(x)
       choose try where z:=(s1) x
                   where result:=() r1(z)
              try where z:=(s2) x
                   where result:=() r2(z)
       end
  end
end
```

Ici, en cas d'échec de la première branche, la deuxième est inspectée sans avoir à recalculer la valeur de `y`.

1.5 Opérateurs Associatifs et Commutatifs

Une autre caractéristique d'ELAN est de permettre au programmeur d'utiliser des opérateurs associatifs et commutatifs (notés AC). Ces opérateurs sont binaires et ont comme première particularité de ne pas imposer une place fixe à leurs arguments (c'est la commutativité). L'autre particularité dit que lorsqu'un même opérateur associatif-commutatif apparaît plusieurs fois dans une expression, il n'y a pas de priorité particulière pour en évaluer un plutôt qu'un autre (c'est l'associativité).

Plus formellement, un symbole $f_{AC} \in \mathcal{F}$ est dit associatif-commutatif s'il satisfait les deux axiomes suivants :

$$\forall x,y,z \in \mathcal{X}, f_{AC}(x,f_{AC}(y,z)) = f_{AC}(f_{AC}(x,y),z) \quad \text{et} \quad f_{AC}(x,y) = f_{AC}(y,x).$$

En ELAN, de tels opérateurs se déclarent en utilisant l'attribut `(AC)`. Considérons, par exemple, la signature d'un module définissant les polynômes sur les entiers :

```
operators global
  X             :                 Variable;
  Y             :                 Variable;
  @             : (Variable)      Poly;
  @             : (int)           Poly;
  @ + @         : (Poly Poly)     Poly assocRight pri 1 (AC);
  @ * @         : (Poly Poly)     Poly assocRight pri 2 (AC);
  deriv(@,@)    : (Poly Variable) Poly;
end
```

En utilisant une telle signature, les expressions `3*X*X+2*X+1` et `X*2+1+X*3*X` sont des termes de sorte `Poly` et correspondent au même polynôme $3X^2 + 2X + 1$ (en tant qu'objet mathématique). C'est grâce à l'associativité et à la commutativité des opérateurs `*` et `+` que les expressions `3*X*X`, `X*3*X` et `X*X*3` correspondent au même monôme $3X^2$ et que les différentes possibilités pour additionner les monômes 1, $2X$ et $3X^2$ mènent toutes au même résultat : le polynôme $3X^2 + 2X + 1$.

Soient s et t deux termes, on écrit $s =_{AC} t$ pour indiquer qu'ils sont égaux *modulo* les axiomes d'associativité et de commutativité. En reprenant l'exemple précédent on a bien :

$$(3 * X * X) + (2 * X) + 1 =_{AC} (X * 2) + 1 + (X * 3 * X)$$

Nous avons vu que le mécanisme d'évaluation d'ELAN repose sur la réécriture. Cela consiste à trouver une règle dont le membre gauche filtre vers le sujet puis à appliquer cette règle pour construire le terme réduit. Mais lorsque le membre gauche de la règle contient un symbole AC la notion de filtrage présentée en 1.2 doit être étendue modulo les axiomes d'associativité et de commutativité.

Afin de mieux comprendre ou sentir les difficultés sous-jacentes au filtrage associatif-commutatif, considérons les simplifications qui permettent d'éliminer l'addition d'une constante nulle et de réduire les polynômes multipliés par 0 ou 1. Pour décrire cela en ELAN, il suffit de définir les 3 règles de réécriture suivantes :

```
rules for Poly
  P : Poly;
global
  []  0+P => P end
  []  0*P => 0 end
  []  1*P => P end
end
```

Considérons maintenant le terme `X*1*3*X`. Sans vous en apercevoir, vous venez de réaliser un grand nombre de transformations qui font que l'objet que vous avez en tête n'est peut être plus `X*1*3*X` mais $3X^2$. Cela provient de votre entraînement et facilité à manipuler des polynômes qui font que vous avez inconsciemment regroupé les X, éliminé le facteur multiplicatif 1 et permuté la variable X avec l'entier 3 pour obtenir une représentation « conventionnelle ».

Plaçons nous maintenant dans le cadre d'ELAN qui est un langage destiné à être exécuté sur un ordinateur ne possédant aucun goût particulier pour la manipulation des polynômes. Le terme

`X*1*3*X` doit donc être simplifié en utilisant seulement les règles définies dans le programme : il s'agit donc de trouver une règle `l => r` et un filtre σ tels que `l`$\sigma =_{AC}$ (`X*1*3*X`) (il faut noter ici l'utilisation de l'égalité modulo AC : $=_{AC}$).

Considérons maintenant la troisième règle `1*P => P` et la substitution qui associe le terme `X*3*X` à la variable `P`, nous pouvons alors remarquer que cette règle peut s'appliquer sur le terme `X*1*3*X` pour le simplifier en `X*3*X`.

Toute la difficulté du filtrage AC consiste à trouver de telles substitutions parce qu'il faut prendre en compte les différentes manières d'associer et de permuter les éléments qui composent le membre gauche de la règle et le terme à réduire. En contre-partie, l'expressivité des règles de réécriture est accrue, ce qui a pour principal effet positif de diminuer le risque d'erreur de la part du programmeur et d'améliorer considérablement la sûreté et la qualité des spécifications ainsi écrites. Étant donnée la complexité des algorithmes de filtrage modulo AC, il est clair que l'utilisation de symboles AC dans une spécification entraîne nécessairement une baisse générale des performances du système en terme de nombre de règles de réécriture appliquées par seconde. En revanche, lorsqu'un même problème est spécifié une fois en utilisant des symboles AC et une autre fois sans en utiliser, il n'est pas aussi évident de savoir quelle spécification s'exécutera le plus rapidement. Un des objectifs de cette thèse est principalement de montrer que des techniques de compilation particulières permettent d'utiliser des symboles AC sans craindre de voir les performances diminuer dramatiquement par rapport à une spécification équivalente ne possédant pas de symbole AC. Le principal intérêt est d'inciter les programmeurs à utiliser des symboles AC dans leurs spécifications, afin d'améliorer la qualité du code ainsi écrit.

1.6 Modularité

La spécification d'un programme relativement complexe n'est jamais une chose aisée. Elle peut cependant être facilitée si l'expressivité du langage de spécification utilisée est grande. Plusieurs caractéristiques du langage ELAN visent à aider le programmeur à écrire le plus facilement possible des spécifications correctes :

- la flexibilité offerte par l'utilisation de grammaires hors contextes pour définir la syntaxe et la structure des données. Elle permet en effet de réduire l'écart entre les notations habituellement utilisées en mathématiques et celles utilisées pour programmer un algorithme.
- la simplicité d'utilisation des règles non nommées permet d'exprimer facilement des opérations de simplification d'expressions ou des fonctions d'accès aux structures de données.
- la puissance des stratégies permet de mieux coordonner les différentes phases de calcul d'un algorithme tout en séparant de manière claire la notion de réduction de la notion de contrôle.

Si l'expressivité d'ELAN se limitait à ces trois points, le langage ne serait pas agréable à utiliser : il serait en effet pénible de devoir tout spécifier à chaque fois et de ne pas disposer d'un mécanisme permettant de réutiliser des morceaux de spécifications écrits par d'autres personnes.

La notion de *modularité* permet de diviser une spécification en entités logiques appelées *modules*. Chaque module peut définir des sortes, des opérateurs, des règles ou des stratégies et importer d'autres modules si besoin est. Précédemment, pour définir la fonction factorielle, nous avons supposé qu'il existait un module définissant la sorte `int` et les opérations usuelles définies sur les entiers. Il a alors suffi d'importer ce module pour disposer de son contenu.

Dans un premier temps, l'importation d'un module peut se voir comme une copie textuelle du contenu du module. Il est cependant intéressant, pour améliorer la qualité de la spécification, de

pouvoir définir des opérateurs « cachés » qui ne peuvent pas être importés par d'autres modules. Cela permet, entre autres, d'*encapsuler* des structures de données et de voir les autres modules comme des clients potentiels tout en s'assurant qu'il ne pourront pas accéder aux structures internes du module.

C'est l'utilisation du mot clé `local` qui permet de déclarer qu'un opérateur, une règle ou une stratégie ne seront pas exportés et resteront invisibles aux autres modules. L'attribut `global` permet au contraire de rendre accessible aux autres modules la définition d'un opérateur, d'une règle ou d'une stratégie. Mais que devient un opérateur exporté ? Est-il local ou global au module qui l'importe ?

Ce petit manque de précision se règle en spécifiant, lors de l'importation d'un module, si les opérateurs importés sont eux-mêmes exportables ou non. Une importation globale signifie donc que tout ce qui est importé est réexporté vers les autres modules, alors qu'une importation locale cache les opérateurs importés.

La notion de module permet de définir les « briques » qui composent un projet. Il est par ailleurs fréquent qu'un grand nombre de « briques » se ressemblent sans être parfaitement identiques : en effet, cela arrive lorsque les modules sont construits en suivant un même processus de fabrication où seulement quelques paramètres sont changés. On parle alors de *modules paramétrés*.

L'environnement ELAN permet de définir des modules paramétrés. Il est ainsi possible de spécifier le module « liste de quelque chose » où *quelque chose* peut être remplacé par un nom de sorte :

```
module list[X]
  import int;     end
  sort X list[X]; end

  operators global
    nil       :               list[X];
    cons(@,@) : ( X list[X] ) list[X];
    size(@)   : ( list[X] )   int;

  rules for int
    e : X;
    l : list[X];
  global
    [] size(nil) => 0         end
    [] size(e.l) => 1+size(l) end
  end
end
```

L'exemple précédent définit le module `list` paramétré par `X`, où `X` peut être remplacé par `int` ou `term`, par exemple, pour définir des listes d'entiers ou des listes de termes. En ELAN, le mécanisme d'intanciation des modules est assez simple : lors de l'analyse syntaxique, les valeurs associées aux paramètres sont connues (`X=term` par exemple) ; avant d'analyser le contenu d'un module paramétré, les paramètres sont remplacés par leur valeur dans le corps du module. Considérons par exemple le cas où le module `list[term]` est importé, il faut en fait imaginer que c'est le module instancié de `list[X]` qui est effectivement importé :

```
module list[term]
  import int;     end
  sort term list[term]; end

  operators global
    nil       :                     list[term];
    cons(@,@) : ( term list[term] ) list[term];
    size(@)   : ( list[term] )      int;

  rules for int
    e : term;
    l : list[term];
  global
    [] size(nil) => 0         end
    [] size(e.l) => 1+size(l) end
  end
end
```

Chapitre 2

Outils pour spécifier et programmer

2.1 Bibliothèque 27
2.2 Parseur 29
2.3 Interpréteur 30
2.4 Compilateur 32
2.5 Comparaison avec d'autres environnements de spécification 33

La création de logiciels informatiques a, par le passé, été considérée comme un art réservé. Un des objectifs des chercheurs travaillant dans le domaine des spécifications algébriques est d'offrir aux informaticiens un moyen d'exprimer « clairement » les comportements attendus d'un logiciel.

D'un point de vue purement théorique, seule la définition du langage de spécification est nécessaire pour être capable d'écrire des spécifications. Il est cependant très difficile, voire impossible, d'écrire des spécifications correctes lorsqu'aucun outil informatique n'est disponible : comment écrire des spécifications de haut niveau si les opérations élémentaires, de bas niveau, doivent être spécifiées à chaque fois ? Comment vérifier que la syntaxe utilisée est bien correcte ? Comment expérimenter le comportement des algorithmes s'ils ne peuvent pas être exécutés ? Comment rendre exécutables et utilisables les spécifications écrites ?

C'est précisément le rôle d'un environnement de spécification que de mettre à disposition du programmeur des outils capables de l'aider à écrire des spécifications. Le cadre logique ELAN dispose ainsi de quatre composants principaux dont le but est de répondre aux questions précédentes : la bibliothèque met à disposition du programmeur des éléments largement réutilisables. Le parseur permet de vérifier la syntaxe des spécifications. L'interpréteur est un outil interactif permettant d'étudier le comportement des algorithmes. Le compilateur est un outil qui transforme des spécifications abstraites en des programmes exécutables indépendants pouvant s'intégrer dans des réalisations logicielles plus importantes.

2.1 Bibliothèque

L'aspect modulaire d'ELAN permet d'écrire des morceaux de spécification qui peuvent être réutilisés pour développer des spécifications plus complexes. Certains modules, à caractère général, sont ainsi regroupés pour former une *bibliothèque*.

Le langage de spécification ELAN permet de définir une grande variété de types de données et d'opérations, mais dans certains cas, il est nécessaire d'intégrer un type de donnée ou une

opération particulière au langage lui-même. Dans le cadre de la réécriture conditionnelle, par exemple, il faut que la notion de satisfaisabilité d'une condition fasse partie intégrante du langage de spécification. Les langages tels que Maude (Clavel, Durán, Eker, Lincoln et Meseguer 1998) ou ASF (Klint 1993, Deursen, Heering et Klint 1996) permettent de décrire des règles conditionnelles où les conditions sont de la forme `if t=s`. Une telle condition est alors satisfaisable lorsque les termes `t` et `s` sont *égaux*. La notion d'égalité doit alors faire partie du système. En ELAN, une approche légèrement différente a été choisie : les conditions sont de la forme `if c` où `c` est un terme de la sorte `bool`. Une telle condition est satisfaisable si le terme `c` se simplifie en la constante `true`. Ici, ce n'est plus la notion d'égalité, mais la valeur `true` et la sorte `bool` qui doivent faire partie intégrante du système. La sorte `bool` et les opérateurs `true` sont alors dit *élémentaires* ou *builtins*.

Pour des raisons de simplicité et d'efficacité, un certain nombre de modules définissant des sortes et opérations élémentaires sont intégrés au système :

- `ident` : la sorte `ident` permet de représenter des listes de caractères alphabétiques. Par exemple, les identificateurs `a`, `b`, `ab`, *etc.* sont des éléments composant la sorte `ident`.
- `bool` : comme mentionné précédemment, la sorte `bool` permet de représenter les valeurs de vérité `true` et `false`, utilisées lors de l'évaluation des conditions, ainsi que certaines opérations élémentaires telles que la conjonction, la disjonction ou la négation (`and`, `or` et `not`).
- `builtinInt` : la sorte `builtinInt` permet de représenter les entiers signés ainsi que les opérations usuelles (addition, soustraction, multiplication, *etc.*).
- `builtinString` : la sorte `builtinString` permet de représenter des chaînes de caractères (une suite de caractères ASCII comprise entre des doubles quotes '`"`').
- `builtinStdio` : ce module offre la possibilité au système d'effectuer des entrées/sorties.
- `cmp` : ce module est particulier dans la mesure où il est paramétré par un nom de sorte s. Il définit alors des opérations de comparaison (égalité, diségalité) pour la sorte s.
- `occur` : ce module est paramétré par deux sortes s_1 et s_2 et définit une opération binaire `occurs in` indiquant si le terme de sorte s_1 passé en premier argument est un sous-terme du deuxième argument de sorte s_2. Considérons les termes $g(a)$ et $f(g(a))$, l'expression `occurs g(a) in f(g(a))` se réduit en `true` parce que `g(a)` apparaît dans le terme `f(g(a))`.
- `replace` : ce module est aussi paramétré par deux sortes et définit l'opération de remplacement sur les termes. Considérons le terme `replace a by b in f(a)`, après réduction, le terme se réécrit en `f(b)`.

Afin de faciliter l'écriture des spécifications, un certain nombre de structures de données ont été spécifiées en ELAN et intégrées au système sous forme de bibliothèque. Cela évite aux programmeurs d'avoir à redéfinir des types de données fréquemment utilisés lors de chaque utilisation. Sont ainsi définis, dans cette deuxième couche composant la bibliothèque, des modules permettant de manipuler des listes, des tuples ou des tableaux. D'autres structures de données, plus spécifiques aux domaines pour lesquels le système ELAN est destiné, sont aussi définies. La bibliothèque contient ainsi des modules permettant de définir et manipuler des termes, substitutions, contraintes et des systèmes équationnels par exemple. Cette liste n'est pas exhaustive dans la mesure où le système évolue et s'enrichit tout au long de sa vie.

Depuis peu, le langage de stratégies a été étendu (Borovanský 1998) afin de le rendre plus expressif et de permettre à l'utilisateur de définir des stratégies paramétrées. De nouvelles fonctionnalités permettant de créer dynamiquement des stratégies ont aussi été ajoutées au système et se présentent sous forme de bibliothèques écrites en ELAN. Cette troisième couche de la bibliothèque regroupe ainsi un ensemble de modules permettant d'accéder au « nouveau » langage

de stratégie. Le lecteur est invité à se référer à la thèse de Peter Borovanský (1998) pour obtenir plus d'informations concernant les fonctionnalités et les techniques d'implantation de cette version étendue du langage de stratégie.

2.2 Parseur

Dans tout environnement de programmation, le parseur est un élément essentiel. C'est d'une part l'outil qui permet de vérifier si le programme écrit est syntaxiquement correct, mais dans de nombreux cas, des phases d'analyse statique du programme sont aussi intégrées afin de déceler d'éventuelles erreurs de typage.

Un parseur n'est généralement pas un outil monolithique mais est au contraire constitué d'une multitude de couches ayant un rôle bien précis. On peut noter parmi celles-ci :

- l'*analyse lexicale* doit décomposer la suite de caractères constituant un programme source en unités lexicales (appelée *lexèmes*) qui sont les briques de base de la structure d'un programme telles que les mots clés, les chaînes de caractères ou les entiers ;
- l'*analyse syntaxique* obtient une suite de lexèmes en entrée et doit trouver dans cette séquence la structure du programme. Dans le cadre d'un langage fondé sur la réécriture, les unités syntaxiques sont les variables, les termes, les règles et les stratégies, entre autres ;
- l'*analyse sémantique* a pour but de vérifier certaines propriétés fondamentales qui ne peuvent pas être décrites à l'aide d'une grammaire hors contexte. Savoir si une variable est bien déclarée ou s'assurer d'une certaine cohérence des types par exemple. Cette phase du parseur reçoit donc le programme sous une forme abstraite (il s'agit souvent d'un *arbre abstrait*) et calcule des propriétés appelées aussi *décorations*. Ces propriétés, telles que la visibilité des opérateurs, sont ajoutés à l'arbre abstrait pour le *décorer*.

Dans le cadre d'ELAN, la phase d'analyse syntaxique est un peu plus complexe que celles habituellement rencontrées dans les autres langages de programmation. Il n'est pas possible d'utiliser des générateurs d'analyseurs lexicaux et syntaxiques tels que *Lex* et *Yacc* (Lesk 1975, Aho, Sethi et Ullman 1989, Wilhelm et Maurer 1994). Il y a bien sûr une partie de la syntaxe d'ELAN qui est suffisamment « figée » pour être traitée par un outil tel que Yacc, mais la grande difficulté vient de la possibilité de définir des opérateurs infixés : la syntaxe des opérateurs est donnée dans une spécification ELAN elle-même. Il faut ainsi construire dynamiquement un analyseur, dépendant de ces règles de grammaire hors contexte, pour être capable de lire et reconnaître la suite de la spécification. C'est l'algorithme d'Earley (1970) qui est utilisé pour analyser les morceaux du programme qui dépendent des règles de grammaire hors contexte. Le reste étant analysé par un automate généré par un outil comparable à Yacc. La partie frontale d'ELAN commence à se dessiner et s'organise autour d'une coopération étroite entre l'analyseur lexical, l'automate d'analyse syntaxique et l'analyseur fondé sur l'algorithme d'Earley. La complexité du parseur d'ELAN ne s'arrête pas là, elle doit en effet son existence à la présence d'un préprocesseur relativement original. Rassurez-vous, l'objectif de cette partie n'est pas d'expliquer en détail comment le parseur actuel est implanté, mais plutôt d'expliquer son fonctionnement général pour mettre en lumière les difficultés rencontrées et aider à concevoir une nouvelle architecture d'environnement de spécification.

Dans bon nombre de langages, le préprocesseur est un outil relativement simple qui intervient avant la phase d'analyse pour y effectuer des remplacements purement syntaxiques. Le préprocesseur d'ELAN ne se limite pas à effectuer des remplacements syntaxiques et doit être vu comme un *générateur de programmes*. Il offre une construction `FOR EACH v SUCH THAT v:=()e : { s }`,

qui remplace dans la suite de lexèmes `s` toutes les occurrences de la variable `v` par une forme normale du terme `e`. On peut ainsi écrire le morceau de spécification suivant :

```
operators global
  FOR  EACH L:list[identifier] AND F:identifier
  SUCH THAT L:=() a.b.nil AND F:=(listExtract) extract(L) :
    { F : term; }
end
```

Ici, `listExtract` est une variante (pour la sorte `list[identifier]`) de la stratégie que nous avons définie au paragraphe 1.4. La construction `FOR EACH` précédente va donc extraire les éléments `a` et `b` de la liste `a.b.nil` pour créer les règles de grammaire hors contexte :

```
operators global
  a : term;
  b : term;
end
```

Le préprocesseur peut donc être utilisé pour générer automatiquement des éléments de spécification qui sont utiles pour analyser la suite du programme. Il faut aussi noter que le préprocesseur a besoin de toute la puissance de l'interpréteur pour effectuer ses remplacements : il doit pouvoir exécuter la stratégie `listExtract` pour être capable d'analyser la suite du module. Ceci renforce encore l'interaction existante entre les différentes phases de la partie frontale d'ELAN : le préprocesseur a besoin de l'interpréteur, l'interpréteur a besoin de l'analyseur syntaxique qui a lui même besoin du préprocesseur. À cela s'ajoute un module de transformation de programmes qui est nécessaire pour rendre exécutable le méta-langage de stratégie défini dans (Borovanský 1998).

Autant dire que le problème est complexe et que le parseur actuel s'apparente de plus en plus à un outil monolithique tant redouté par les informaticiens. Pour répondre à ces craintes et conserver toute la puissance du préprocesseur actuel, différents *scenarii* ont été envisagés. Le chapitre 3 de ce document traite de l'organisation interne d'un environnement de spécification et propose différentes solutions pour modulariser l'architecture de l'environnement ELAN, en particulier le fonctionnement du parseur et du préprocesseur.

2.3 Interpréteur

À l'image d'un dialecte humain, qui ne peut survivre que s'il est parlé, un langage de spécification ne sera utilisé que s'il s'appuie sur des outils spécifiques. L'interpréteur fait partie des outils qui donnent un sens aux spécifications écrites : il permet d'évaluer les expressions bien formées (reconnues par le parseur) en « interprétant » les constructions élémentaires. Dans le cadre d'ELAN, l'évaluation d'une spécification se fait en fonction d'un terme d'entrée appelé *requête* ou *query*. Étant donné un terme clos, le calcul d'une de ses formes normales consiste à appliquer successivement les règles et les stratégies définies dans la spécification. Une des caractéristiques de l'interpréteur est d'évaluer la spécification au fur et à mesure, sans effectuer un travail de préparation préalable trop important.

En donnant un sens aux spécifications, l'interpréteur fait du langage de spécification une entité concrète et observable. En particulier, le non-déterminisme inhérent au langage perd son côté magique : il devient complètement cerné et modélisé. Au paragraphe 1.3 nous parlions de double non-déterminisme : le choix de la règle à appliquer et la position dans le terme où

appliquer la règle. Le choix d'appliquer les règles en utilisant une stratégie de *leftmost-innermost* permet de réduire à un le niveau de non-déterminisme : la stratégie fixe la position où les règles s'appliquent dans le terme. Reste le choix de la règle à appliquer, qui est en partie guidé par la position, puisqu'il faut que le symbole de tête du membre gauche soit le même que celui se trouvant à la position choisie, mais il peut exister plusieurs règles satisfaisant ce critère. Dans ce cas l'interpréteur sélectionne la première règle dans sa liste de règles commençant par un symbole donné. On parle alors de réécriture ordonnée, puisque les règles commençant par un même symbole de tête sont toujours appliquées dans le même ordre. Mais il faut noter que ce n'est absolument pas une propriété du langage de spécification : rien n'indique que les évolutions futures du système satisferont cette propriété.

Le même phénomène se produit avec l'opérateur $\mathsf{dc}(s_1, \ldots, s_n)$ qui devrait choisir aléatoirement une stratégie sans échec parmi $s_1, \ldots, s_n$. Dans la pratique, c'est l'opérateur first qui est implanté, mais une fois encore, ce n'est pas une propriété du langage initial. Il existe d'ailleurs une extension concurrente d'ELAN (Borovanský et Castro 1998) qui exécute en parallèle les sous-stratégies $s_1, \ldots, s_n$ et sélectionne la première qui termine sans échec.

Lorsque l'implantation en C++ de l'interpréteur a débuté, l'objectif initial n'était pas de construire l'interpréteur le plus efficace possible, c'est pourquoi il ne se démarque pas de ses concurrents en terme d'efficacité. Son utilisation reste néanmoins agréable et peu limitée. Les techniques utilisées pour implanter les différents composants sont relativement simples et offrent un bon compromis entre la facilité de mise en œuvre et la vitesse d'exécution.

Étant donné un terme clos, pour savoir quelles sont les règles qui peuvent s'appliquer, une première sélection est faite en fonction du symbole de tête du terme : seules les règles dont le membre gauche a le même symbole de tête sont retenues. Les règles sont ensuite essayées l'une après l'autre. Ce schéma général s'applique aussi pour les règles qui contiennent des opérateurs Associatifs et Commutatifs, mais il faut savoir que l'algorithme de filtrage n'est pas complètement intégré à l'interpréteur ELAN. Après son doctorat, Steven Eker a développé et implanté un algorithme de filtrage AC (1995) suffisamment efficace et stable pour être réutilisé par d'autres logiciels. L'idée de confier tous les problèmes de filtrage AC à cet outil a donc été retenue lors de la réalisation de l'interpréteur ELAN. C'est évidemment un bon choix en termes de simplicité, de vitesse de développement et de sûreté du logiciel, mais plus contestable en ce qui concerne l'efficacité du produit obtenu. Sans entrer dans les détails, simplement parce que le reste de ce document devrait donner quelques pistes permettant d'améliorer l'efficacité du filtrage AC en général, on peut souligner les deux points qui paraissent être responsables de l'inefficacité de l'interpréteur lorsque des symboles AC sont utilisés :

- L'algorithme de filtrage AC développé par Steven Eker (1995) est un outil indépendant. Cela signifie qu'il travaille sur ses propres structures de données. Lors de chaque tentative d'application d'une règle, il faut donc convertir le membre gauche de la règle et le terme clos (qui sont codés avec les structures de données d'ELAN) vers la structure de données de l'outil de filtrage. Il faut ensuite effectuer la conversion inverse pour pouvoir récupérer les solutions du problème de filtrage et les utiliser pour appliquer la règle. Dans la pratique, le coût de cette double conversion est largement supérieur au temps passé dans la procédure de filtrage proprement dite.
- Le deuxième goulot d'étranglement est aussi lié à un problème de conversion : pour des raisons de simplicité et d'efficacité, les algorithmes de filtrage utilisent une représentation particulière des termes, dite *aplatie*, où les occurrences multiples d'un même symbole AC sont éliminées et les sous-termes sont triés. Pour être parfaitement indépendant, l'outil de filtrage calcule donc cette forme « aplatie » avant chaque étape de résolution. Cette

deuxième phase de transformation est elle aussi très coûteuse en temps de calcul et pourrait être évitée si l'intégration était meilleure.

La complexité théorique d'un algorithme de filtrage AC est sans commune mesure avec celle du filtrage syntaxique. Si à cela s'ajoutent des problèmes pratiques qui rendent le temps de résolution des problèmes AC inférieurs aux temps de conversion, il devient clair que des problèmes de performances apparaissent. Dans le cadre de la réalisation d'un prototype, l'importance est moindre et l'essentiel est que cela permette de résoudre des problèmes de filtrage difficiles et de pouvoir appliquer des règles de réécriture modulo l'associativité et la commutativité.

Pour terminer cette brève description de l'interpréteur, il faut savoir que lorsqu'une règle s'applique, les évaluations locales sont calculées, puis le terme résultant est construit. Dans l'implantation courante, le membre droit de chaque règle est partiellement pré-construit en mémoire : il s'agit d'un terme qui n'est pas complètement bien formé puisqu'il contient des « trous » correspondant aux variables dont la valeur n'est pas connue avant l'application de la règle. Pour construire le terme réduit, il suffit donc de dupliquer le « guide » pré-construit en mémoire et de compléter les « trous » par les instances des variables qui sont calculées par l'étape de filtrage. Une fois le nouveau terme construit, avant d'essayer de le réduire à nouveau, il faut libérer l'espace mémoire qui était occupé par le terme précédent. La gestion mémoire est faite en utilisant des « compteurs de références », ce qui permet de savoir si un terme est partagé ou non en mémoire : lorsqu'un terme n'est plus utilisé, le compteur indique qu'il y a zéro référence vers le terme et la place qu'il occupe en mémoire peut être libérée.

2.4 Compilateur

Un compilateur est aussi un outil permettant de donner un sens aux spécifications en les rendant exécutables. À la différence d'un interpréteur, un compilateur ne fait que traduire d'un langage source en un langage cible. L'objectif n'est plus d'évaluer les expressions du langage source, en les interprétant, mais de les traduire en des expressions équivalentes exprimées dans un autre langage. Pour que la spécification initiale devienne exécutable, il faut que le langage cible dispose d'outils permettant de l'exécuter : ici encore, il peut s'agir d'un interpréteur ou d'un compilateur.

Définir une frontière nette entre interprétation et compilation n'est jamais très facile à faire parce que beaucoup de concepts différents se cachent derrière les termes « compilateur », « interpréteur », « compilation », « interprétation », « langage compilé » et « langage interprété ». Le chapitre 4 devrait aider à clarifier la situation.

Sans vouloir alimenter de polémique, les techniques de compilation ont, d'une manière générale, deux principaux atouts par rapport aux interpréteurs :

- ce n'est pas une vérité absolue, mais habituellement, pour un langage source donné, les compilateurs permettent d'obtenir une implantation plus efficace ;
- le deuxième avantage est de produire des exécutables qui deviennent « indépendants » de l'environnement de développement : les exécutables peuvent être utilisés seuls par d'autres outils, sans nécessiter la présence d'un interpréteur. Dans certains cas, le code cible généré par le compilateur peut même être directement utilisé et intégré dans le développement d'un logiciel plus complexe.

Étant donnée une spécification ELAN, le compilateur doit permettre de la traduire en un autre programme dont le comportement est équivalent au premier. L'efficacité du programme obtenu doit être suffisante pour satisfaire l'utilisateur : le compromis entre l'expressivité du langage de spécification, la facilité d'utilisation, le temps de développement et le temps d'exécution doit être

« bon ». D'un point de vue pratique, la fiabilité et la vitesse d'exécution du programme obtenu doit permettre de développer des outils qui peuvent être réutilisés pour résoudre des problèmes difficiles et même être intégrés à l'environnement de spécification ELAN.

2.5 Comparaison avec d'autres environnements de spécification

Il existe un grand nombre d'outils liés à la réécriture, mais une grande partie de ces systèmes sont des logiciels de déduction automatique qui utilisent la réécriture de façon interne pour réduire et normaliser des termes. Nous nous intéressons ici aux caractéristiques des principales réalisations logicielles fondées sur la logique de réécriture (le lecteur intéressé par une comparaison des différents formalismes de spécification peut se reporter au *survey* de Martin Wirsing (1995) pour plus de précisions). Dans ces logiciels, la logique de réécriture n'est pas seulement une technique interne de résolution, mais le paradigme principal de calcul offert à l'utilisateur. C'est pourquoi nous ne retenons que les quatre environnements suivants : ASF, CafeOBJ, Maude et OBJ-3.

OBJ-3. Ce système (Goguen et al. 1987) est particulier dans la mesure où il a été conçu en 1986 au SRI. On ne peut plus dire qu'il soit maintenu, mais il a été le précurseur en termes d'idées et de conception de la plupart des autres systèmes existant à ce jour. L'histoire de la « saga » OBJ (Goguen 1988a) a commencé en 1976 lorsque Joseph Goguen a défini une version originelle (Goguen 1977) qui était un langage pour des algèbres d'erreurs. La première implantation OBJ-0 était mono-sortée (Goguen 1978, Goguen et Tardo 1977) et date de 1979. En 1983, la version OBJ-1 a été étendue à la réécriture modulo l'associativité et la commutativité (Goguen, Meseguer et Plaisted 1982). En 1985, une nouvelle version fondée sur les algèbres avec sortes ordonnées (Futatsugi, Goguen, Jouannaud et Meseguer 1985, Futatsugi, Goguen, Meseguer et Okada 1987, Futatsugi, Goguen, Jouannaud et Meseguer 1984) a été développée pour mener à OBJ-2. La dernière version OBJ-3 ressemble syntaxiquement à OBJ-2 mais est basée sur une approche plus simple de la réécriture avec sortes ordonnées.

Dans OBJ-3, les signatures utilisées sont avec sortes ordonnées et les systèmes d'équations utilisés pour réduire un terme sont appliqués modulo les axiomes d'associativité et de commutativité. Le système permet aussi de définir des modules paramétrés (Futatsugi et al. 1987, Jouannaud, Kirchner, Kirchner et Mégrelis 1992) et des expressions complexes de modules.

Concernant l'implantation, le système OBJ-3 se compose d'un interpréteur écrit en Common Lisp qui permet d'interfacer les spécifications comprenant des règles de réécriture avec des fonctions écrites en Lisp. Il existe par ailleurs une autre implantation d'OBJ-3 écrite en C (Cavenaghi, de Zanet et Mauri 1987).

La famille des systèmes OBJ a été utilisée assez longtemps pour prototyper des idées et son intérêt a été montré par le grand nombre de spécifications écrites dans ce formalisme (Battiston, de Cindio et Mauri 1988, Collavizza 1989, Collavizza et Pierre 1988, Goguen 1988b, Stavridou 1988, Eker 1991, Nakagawa, Futatsugi, Tomura et Shimizu 1987, Christopher 1988).

Maude. Ce système, développé au SRI par l'équipe de José Meseguer (Clavel et al. 1998), est lui aussi fondé sur la logique de réécriture. Il intègre actuellement les paradigmes de programmation fonctionnelle et objet. Sa sémantique est fondée sur la *logique équationnelle d'appartenance* introduite dans (Meseguer 1998, Bouhoula, Jouannaud et Meseguer 1997). Cette logique, semblable à celle développée dans (Hintermeier, Kirchner et Kirchner 1994, Hintermeier, Kirchner et Kirchner 1995), est une extension conservative de la logique équationnelle avec sortes ordonnées

et de la logique équationnelle partielle. Elle permet en particulier le sous-typage, la définition d'opérateurs partiellement définis et la surcharge d'opérateurs. Les formules atomiques de cette logique sont des équations conditionnelles de la forme $t = t'$ et des assertions d'appartenance, notées $t : s$, signifiant que le terme t doit appartenir à la sorte s.

Les déclarations d'opérateurs et de sous-sortes sont vues comme des axiomes d'appartenance. Considérons par exemple la sorte *Entier* et sa sous-sorte *Naturel* (tous les *Naturels* sont des *Entiers*). Considérons une fonction f définie sur les *Entiers* dont les valeurs sont des *Naturels*. La définition d'une telle fonction peut s'exprimer en utilisant deux axiomes d'appartenance :

$$\begin{array}{ll} x : Entier & \textbf{if} \quad x : Naturel, \text{ et} \\ f(x) : Naturel & \textbf{if} \quad x : Entier \end{array}$$

À l'image d'ELAN, le système Maude permet de définir des modules fonctionnels, mais il permet en plus de définir des modules orientés objet. Considérons par exemple la sorte *Compte* composée entre autres du *montant* disponible et du *nom* du propriétaire d'un compte bancaire. L'approche orientée objet permet d'écrire des règles de transformation d'états dans lesquelles il n'est pas nécessaire d'exprimer l'ensemble des champs composant la sorte manipulée, ni l'ensemble des objets existant en mémoire. On peut ainsi écrire :

```
transférer M de C1 vers C2
  <C1 : Compte | montant : M1>   <C2 : Compte | montant : M2>
  =>
  <C1 : Compte | montant : M1-M> <C2 : Compte | montant : M2+M>
  if M1 >= M
```

L'ensemble des objets existants est géré de façon interne en utilisant un opérateur associatif-commutatif avec élément neutre (ACI).

Une des particularités de Maude est de permettre d'appliquer efficacement des règles modulo plusieurs théories, à savoir les différentes combinaisons des axiomes d'associativité, de commutativité, d'identité et d'idempotence. Une partie des techniques d'implantation utilisées sont présentées dans (Eker 1996).

Une autre originalité du système Maude est d'exploiter la réflexivité de la logique de réécriture (Clavel et Meseguer 1996, Clavel 1998).

Partant du fait qu'il existe une théorie de réécriture universelle $\mathcal{U}$ permettant d'interpréter toutes les autres théories $\mathcal{TR}$:

$$\mathcal{TR} \vdash t \Rightarrow t' \qquad \textbf{ssi} \qquad \mathcal{U} \vdash \langle \overline{\mathcal{TR}}, \overline{t} \rangle \Rightarrow \langle \overline{\mathcal{TR}}, \overline{t'} \rangle$$

où $\overline{\mathcal{TR}}$, $\overline{t}$, $\overline{t'}$ sont les codages respectifs de la théorie $\mathcal{TR}$ et des termes t,t' dans la théorie universelle $\mathcal{U}$. Le système Maude propose deux sortes élémentaires $Term$ et $Module$ permettant de représenter ces codages ($\overline{t} : Term$ et $\overline{\mathcal{TR}} : Module$) et des primitives de conversion entre un terme $t : s$ et sa représentation codée $\overline{t} : Term$. La théorie universelle $\mathcal{U}$ peut alors être intégrée au système sous forme d'un noyau réflexif. D'un point de vue utilisateur, seules deux primitives principales sont accessibles :

$$\begin{array}{lcl} \textbf{meta-reduce} & : & (Module\ Term) \mapsto Term \\ \textbf{meta-apply} & : & (Module\ Term\ Qid\ Substitution\ Int) \mapsto Term \end{array}$$

La première primitive permet de calculer pour un système de réécriture $\mathcal{TR}$, la forme normale t' d'un terme t : **meta-reduce**$(\overline{\mathcal{TR}}, \overline{t}) = \overline{t'}$ si $\mathcal{TR} \vdash t \Rightarrow t'$. Il faut noter ici que $\mathcal{TR}$ est une donnée de l'environnement d'exécution et que **meta-reduce** lui donne un sens.

La deuxième primitive **meta-apply** offre un plus grand contrôle, parce qu'elle permet d'appliquer une règle spécifiée par son nom, tout en donnant des contraintes sur le filtre à appliquer. L'expression **meta-apply**($\overline{\mathcal{TR}}$,$\overline{t}$,$\overline{\ell}$,$\overline{\theta}$,n) signifie que le terme t doit être réécrit par la règle $\overline{\ell}$ du système $\mathcal{TR}$ et qu'en plus de ces conditions, le filtre trouvé doit satisfaire la substitution θ. L'entier n permet de contrôler l'extraction des solutions du problème de filtrage : seule la $(n+1)$-ième substitution est calculée.

CafeOBJ. C'est un langage de spécification fondé sur trois extensions de la logique équationnelle multi-sortée : la logique équationnelle avec sortes ordonnées, la relation de transition d'états qui permet d'exprimer des systèmes concurrents non-déterministes, et la notion de sortes cachées.

Sa sémantique repose sur une combinaison de la logique de réécriture, des algèbres avec sortes ordonnées et des algèbres avec sortes cachées. D'un point de vue expressivité, les trois extensions proposées ont pour but de rendre CafeOBJ adapté à l'écriture de spécifications algébriques avec sous-typage et de spécifications algébriques concurrentes. Les différentes combinaisons de ses caractéristiques fondamentales (algèbres avec sortes ordonnées, sortes-cachées et la logique de réécriture (Diaconescu 1996)) sont souvent représentées par le « cube CafeOBJ » (Diaconescu et Futatsugi 1996).

Le langage CafeOBJ (Futatsugi et Sawada 1994, Futatsugi et Nakagawa 1996, Futatsugi et Diaconescu 1997) préserve la plupart des caractéristiques du système OBJ-3, à savoir, la syntaxe infixe, le sous-typage, le typage dynamique avec traitement d'erreurs et les modules paramétrés.

Dans ses premières versions, le système CafeOBJ était distribué avec un interpréteur écrit en Common Lisp. Depuis, pour parer à des problèmes de performances relativement médiocres, des techniques de compilation utilisant une machine abstraite ont été développées. Il existe actuellement deux compilateurs indépendants : TRAM (Ogata, Ohara et Futatsugi 1997) et Brute (Ishisone et Sawada 1998). Ce dernier compilateur possède une machine abstraite plus efficace et surtout plus puissante, dans la mesure où elle permet d'effectuer de la réécriture modulo les théories associatives et commutatives par exemple.

ASF+SDF. Ce système se distingue des trois autres dans la mesure où le langage de spécification est de loin le plus simple. Il se compose d'un formalisme de spécification algébrique (ASF) permettant de définir des règles de réécriture conditionnelles et d'un formalisme de définition de syntaxe (SDF) permettant de définir des signatures multi-sortées et des opérateurs associatifs. À l'inverse des autres systèmes, la version actuelle d'ASF+SDF ne permet pas de définir des modules paramétrés et n'offre pas de bibliothèque intégrant des sortes et opérateurs élémentaires (*builtin*).

L'environnement (ou plutôt « méta-environnement ») de spécification a été originellement conçu pour aider au développement de langages de programmation (Deursen et al. 1996), ce qui explique pourquoi l'accent a été particulièrement mis sur les outils d'édition, de développement et d'analyse statique plutôt que sur le développement du formalisme de spécification, qui est largement suffisant pour remplir sa tâche.

ASF+SDF n'est pas un outil monolithique, bien au contraire. Il se compose de plusieurs éléments de base qui communiquent en utilisant un format d'échange commun : l'asFix. Ce format permet de représenter la syntaxe abstraite de n'importe quel objet manipulé. Pour intégrer un nouvel outil à l'environnement, il suffit que celui-ci soit capable de communiquer en utilisant le format asFix. C'est en particulier grâce à cette extrême modularisation que le groupe de Paul Klint a réussi à développer un grand nombre d'outils complexes et relativement stables. Parmi

ceux-ci on compte :

- Un éditeur qui s'adapte automatiquement à la spécification que l'utilisateur exécute (GSE : *Generic Syntax-Directed Editor* (Koorn 1994)). Cet outil permet de fournir, en même temps qu'un logiciel développé dans l'environnement ASF+SDF, un éditeur adapté au logiciel : cet éditeur intègre un module d'analyse syntaxique permettant de vérifier facilement la correction syntaxique des termes donnés en entrée du programme.
- Un analyseur interactif capable de vérifier la syntaxe d'un morceau de texte (appelé *focus*) par simple utilisation de la souris.
- Un outil d'affichage (appelé *pretty-printer*) permettant de mettre en forme les résultats fournis sous forme de termes.
- Un interpréteur doté d'un analyseur incrémental.
- Un compilateur permettant d'effectuer des compilations modulaires.

Le système actuel est en pleine évolution, dans la mesure où une nouvelle organisation interne, appelée ASF+SDF2, est en cours de développement (van den Brand, Heering et Klint 1997, van den Brand, Olivier, Moonen et Kuipers 1997). Dans cette nouvelle architecture, tous les composants sont reliés par un outil de synchronisation et de contrôle fondé sur l'algèbre de processus : le ToolBus (Bergstra et Klint 1995).

Contrairement à ce que laisserait penser la simplicité du langage de spécification, les possibilités offertes par l'environnement n'en sont pas diminuées, comme en témoigne le nombre de développements majeurs réalisés : le compilateur est écrit en ASF+SDF lui-même et compilé en utilisant une technique de *bootstrap*.

ELAN. Décrit dans les chapitres 1, 2 et 3, le système ELAN (Vittek 1994, Borovanský, Kirchner, Kirchner, Moreau et Vittek 1996) est lui aussi fondé sur la logique de réécriture multi-sortée. Résumons ses principales originalités par rapport aux systèmes décrits plus haut.

Il permet de spécifier d'une façon naturelle des procédures non-déterministes, telles que par exemple l'unification modulo différentes théories, la SLD-résolution ou la surréduction.

Une autre particularité du système ELAN est d'intégrer un *préprocesseur* permettant de générer automatiquement des morceaux de spécifications : des blocs de textes génériques peuvent être définis. Au cours de l'analyse syntaxique, des valeurs de variables sont calculées par l'interpréteur ELAN et sont utilisées pour instancier ces blocs de textes.

La construction `FOR EACH` v `SUCH THAT` $v := e : \{\ s\ \}$ remplace la variable v, dans la chaîne s, par tous les résultats obtenus par calcul de formes normales du terme e.

C'était aussi le premier à introduire la notion de stratégie définie par l'utilisateur.

Conclusion. De cette comparaison des différents systèmes existants fondés sur la logique de réécriture, il faut retenir que les formalismes proposés sont relativement proches, même si certains choix théoriques ou pratiques donnent à chacun une originalité particulière :

- les systèmes de la famille OBJ offrent des mécanismes de modularisation et de paramétrisation particulièrement développés ;
- les aspects réflexifs de la réécriture sont bien intégrés dans le système Maude ;
- l'environnement de spécification et la possibilité de traiter des problèmes de taille réelle sont un des points forts du système ASF+SDF ;
- la possibilité d'effectuer des étapes de réécriture modulo un grand nombre de théories sont les points forts des systèmes Maude et CafeOBJ ;

– une des originalités du système ELAN est d'offrir un préprocesseur capable de construire dynamiquement des composants d'un système de calcul au moment de l'analyse syntaxique d'une spécification ;
– enfin, la possibilité de compiler efficacement des applications réelles comprenant des stratégies non-déterministes pour contrôler l'application des règles de réécriture est sans aucun doute un fait marquant qui différencie le système ELAN de tous les autres.

Chapitre 3

Plateforme de prototypage

3.1 Format d'échange 40
3.2 Création d'outils 43
3.3 Système ouvert 45
3.4 Vers une nouvelle architecture 46
3.5 Synthèse 49

Le développement de l'environnement de spécification ELAN est animé par un double objectif :

- ELAN doit être un environnement permettant de prototyper rapidement des outils complexes tout en assurant une certaine qualité et sûreté du logiciel ainsi construit. Le langage de spécification doit donc être suffisamment puissant et expressif pour permettre le développement rapide et la mise au point d'algorithmes complexes. L'environnement doit aussi permettre de rendre « utilisables » les prototypes créés, en assurant une certaine ouverture vers le monde extérieur et une vitesse d'exécution raisonnable.
- La conception d'ELAN doit aussi permettre de mieux comprendre et d'améliorer la qualité des environnements de spécification. Le système ELAN n'est donc pas seulement orienté vers les utilisateurs. Il doit aussi permettre aux concepteurs d'expérimenter facilement de nouvelles idées. Pour cela, sa conception doit permettre l'intégration de nouveaux modules, le remplacement de certains composants et le travail en parallèle de plusieurs chercheurs ou équipes de recherche.

Le premier point est abordé dans cette thèse à travers la *réalisation* d'un compilateur. En effet, celui-ci a pour objectif d'aider l'utilisateur à créer des programmes exécutables qui soient d'une part efficaces, mais aussi indépendants afin de les considérer comme des composants et les intégrer dans un projet de plus grande ampleur.

Le deuxième point est abordé à travers la *conception* du compilateur. En effet, la conception du nouveau composant qu'est le compilateur nous a amené d'une part à étudier et combiner différentes techniques de compilation tout en gardant à l'esprit l'idée de rendre le compilateur le plus extensible possible. D'autre part, cela nous a aussi amené à réfléchir à l'organisation générale de l'environnement de spécification et à proposer des solutions.

Ce chapitre présente l'architecture de l'environnement de spécification ELAN. Celle-ci repose essentiellement sur l'existence d'un format d'échange permettant de modulariser et d'ouvrir notre système vers l'extérieur. Le format d'échange retenu pour ELAN s'est montré bien adapté

au développement d'outils de transformation de programmes par exemple, mais il a cependant le défaut d'être trop proche de la représentation interne des données du système. Profitant de l'expérience du groupe ASF+SDF, nous proposons en fin de chapitre une nouvelle architecture d'environnement fondée sur la définition d'un format plus lisible et plus modulaire.

3.1 Format d'échange

La conception d'un environnement de spécification est une tâche souvent difficile parce qu'elle doit prendre en compte l'intégration de différents outils hétérogènes tels qu'un parseur, un préprocesseur, un interpréteur et un compilateur. Les recherches portant sur l'élaboration d'un bon environnement adapté aux spécifications à base de règles de réécriture sont toujours d'actualité. L'environnement ASF+SDF (Klint 1993) est un bon exemple de conception dans lequel les différents composants sont connectés par un outil appelé ToolBus. Cet outil gère toutes les communications et propose un langage d'échange commun (Bergstra et Klint 1995). L'aspect réflexif de la réécriture fait que, pour être intéressant, un langage de programmation fondé sur la notion de réécriture doit être suffisamment puissant pour permettre d'implanter des outils manipulant des programmes écrits dans le même langage. Dans ce cadre, l'existence d'un format d'échange est clairement reliée au problème de réflexivité dans la mesure où il doit permettre, entre autres, de représenter les programmes par des termes. Une grande partie des problèmes liés à la réflexivité ont été résolus et intégrés dans le langage Maude (Clavel 1998, Clavel et al. 1998). D'une manière générale, l'étude de la coopération de systèmes hétérogènes est un projet ambitieux qui concerne de nombreux champs de l'informatique tels que le Génie Logiciel, l'Intelligence Artificielle, la Déduction Automatique et la Programmation par Contraintes (Dalmas, Gaëtano et Sausse 1996, Dalmas, Gaëtano et Watt 1997, Homann et Calmet 1995). La nécessité d'avoir un format d'échange commun est désormais bien établie.

Il est cependant clair que l'approche consistant à utiliser un format d'échange pour connecter différents composants n'est pas la solution la plus efficace, dans la mesure où des étapes d'encodage et de décodage sont introduites. Il semble cependant que ce soit le prix à payer pour définir un environnement cohérent composé de processus atomiques. De manière analogue à ce qui a été fait dans l'environnement ASF+SDF, nous avons récemment introduit un format d'échange appelé REF (« Reduced Elan Format ») dans l'environnement ELAN.

L'introduction du format REF a été dans un premier temps motivée par le développement du compilateur. Le système était alors relativement monolithique et comprenait déjà un parseur, un préprocesseur et un interpréteur. Il fallait trouver un moyen d'intégrer le nouveau composant qu'est le compilateur. Une solution aurait pu consister à étendre le système existant, avec le risque de voir le système se refermer sur lui-même. Notre objectif était tout autre, puisqu'il consistait à pérenniser les expérimentations faites jusqu'à ce jour. L'introduction d'un format d'échange était une bonne alternative qui avait l'avantage de permettre la réalisation d'un compilateur le plus indépendant possible du système existant et qui permettait aussi la rénovation de l'environnement composant par composant. Nous étions de plus intimement persuadés que ce format d'échange nous offrirait une souplesse et des possibilités supplémentaires.

Le parseur d'ELAN a, dans un premier temps, été étendu afin de permettre l'exportation au format REF de tout programme correctement analysé. L'interpréteur a lui aussi été modifié afin de pouvoir lire et interpréter un programme représenté dans le format REF. Par ce biais, le compilateur pouvait devenir un composant complètement indépendant : il suffisait qu'il soit capable de lire et compiler des programmes codés dans le format REF. La figure 3.1 illustre l'organisation du système actuel.

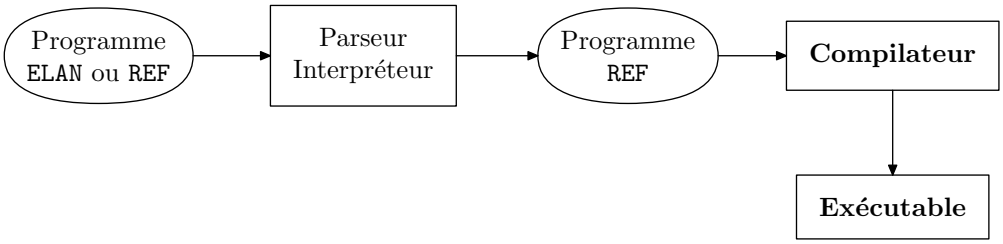


FIG. 3.1 – *Étant donnée une spécification ELAN, le parseur est capable de la lire et de construire une image mémoire interprétable. Cette image peut aussi être exportée au format REF pour être par la suite transformée par le compilateur en un exécutable. Le parseur ELAN est par ailleurs capable de lire une spécification directement écrite dans le format REF. Dans ce dernier cas, la vitesse de chargement du programme est nettement plus élevée, dans la mesure où l'analyse syntaxique du format REF est rendue volontairement simple.*

D'une manière générale, un programme codé dans le format REF peut être considéré comme une représentation aplatie d'un programme ELAN, où toutes les constructions syntaxiques apparaissant dans les différents modules sont fusionnées. La représentation d'une spécification, dans ce format REF, se compose des listes suivantes :

- la liste des identificateurs apparaissant dans le programme ;
- la liste des noms de sortes utilisées dans le programme ;
- la liste des noms de modules composant la spécification ;
- une liste de règles de grammaires pour chaque sorte s, définissant la syntaxe des opérateurs de la spécification ;
- la liste des règles de réécriture définies dans la spécification ;
- la liste des stratégies définies dans le programme.

Dans une première étape, un numéro unique est associé à chaque identificateur apparaissant dans le programme. Ce numéro est ensuite mémorisé dans la première liste du format REF pour être utilisé lors de chaque référence à un identificateur du programme. Considérons par exemple la signature ELAN suivante :

```
module liste
sort Element Liste; end
operators global
  a          :                 Element;
  b          :                 Element;
  nil        :                 Liste;
  @.@        : (Element Liste) Liste;
  extract(@) : (Liste)         Element;
end
```

Son codage dans le format REF s'exprime de la manière suivante :

```
GrammarForSort [Element]:0:
```

```
3:⟨a⟩          :0:8:0:0:0:Ident(⌈a⌉).nil.
3:⟨b⟩          :0:8:0:0:0:Ident(⌈b⌉).nil.
3:⟨extract(@)⟩:0:8:0:0:0:Ident(⌈extract⌉).Char('(').Type(⌈Liste⌉).Char(')').nil.
nil end
GrammarForSort ⌈Liste⌉:0:
3:⟨nil⟩        :0:8:0:0:0:Ident(⌈nil⌉).nil.
3:⟨@.@⟩        :0:8:0:0:0:Type(⌈Element⌉).Char('.').Type(⌈Liste⌉).nil.
nil end
```

Afin d'améliorer la lisibilité, les numéros associés aux sortes `Element` et `Liste` sont notés ⌈`Element`⌉ et ⌈`Liste`⌉ et les numéros associés aux identificateurs `a`, `b`, `nil`, `extract` sont respectivement notés ⌈`a`⌉, ⌈`b`⌉, ⌈`nil`⌉ et ⌈`extract`⌉. Enfin, le code ASCII d'un caractère `c` est noté 'c'.

Chaque règle de grammaire se voit attribuer un nom, noté ⟨`@.@`⟩ par exemple dans le cas de la dernière règle. Les différents paramètres d'une règle de grammaire permettent de coder, respectivement de la gauche vers la droite : la visibilité du symbole (locale ou globale), le nom de la règle, la priorité, des informations syntaxiques (`8` signifie que le symbole est affichable), le statut du symbole (élémentaire ou non), la théorie équationnelle du symbole (actuellement un symbole peut appartenir à la théorie vide ou à la théorie associative et commutative), une information concernant la compilation des stratégies et enfin la liste des unités lexicales qui définissent la syntaxe du symbole. La place d'un argument d'une fonction est représentée par la sorte de cet argument (`Type(⌈Element⌉)` par exemple).

En ELAN, une règle de réécriture est composée principalement de son nom, du membre gauche, du membre droit et d'une liste d'évaluations locales (`if`, `where` ou `choose`).

Considérons l'ensemble de règles ELAN suivant :

```
rules for Element
  element : Element ;
  liste   : Liste ;
global
  [extractrule1] extract(element.liste) => element        end
  [extractrule2] extract(element.liste) => extract(liste) end
end
```

Le codage au format REF du système est le suivant :

```
RULE(
⌈extractrule1⌉,⌈Element⌉,⌈liste⌉,
FSYM(FSYM(VAR(0,⌈Element⌉).VAR(1,⌈Liste⌉).nil, ⟨@.@⟩).nil, ⟨extract(@)⟩,
VAR(0,⌈Element⌉),
nil)

⌈extractrule2⌉,⌈Element⌉,⌈liste⌉,
FSYM(FSYM(VAR(0,⌈Element⌉).VAR(1,⌈Liste⌉).nil, ⟨@.@⟩).nil, ⟨extract(@)⟩,
FSYM(VAR(1,⌈Element⌉).nil,⟨extract(@)⟩),
nil)
```

Les différents paramètres qui composent une règle au format REF sont respectivement de gauche à droite : le nom de la règle, la sorte des termes transformés par la règle, le module dans lequel la règle est définie, le membre gauche, le membre droit et finalement la liste des évaluations

locales (réduite à `nil` dans cet exemple). On peut noter que les noms des variables impliquées dans les règles n'apparaissent pas dans le format REF et sont désignées par un numéro et leur sorte. Dans la première règle, l'expression `VAR(0,⌈Element⌉)` désigne ainsi la variable `element` définie dans la spécification.

Dans le format REF, une stratégie est composée de son nom, de sa sorte et d'une expression construite à partir des constructeurs de stratégies élémentaires présentés dans le paragraphe 1.4. Considérons la définition de stratégie suivante :

```
strategies for Element->Element
implicit
  [] listExtract => iterate*(dc one(extractrule2)) ; dc one(extractrule1) end
end
```

Le codage au format REF correspondant est le suivant :

```
STRATEGY(⌈listExtract⌉,⌈Element⌉,⌈liste⌉,
         iterate(dcone(⌈extractrule2⌉)) ; dcone(⌈extractrule1⌉.nil))
end
```

Dans le format REF, un terme se compose d'une liste de sous-termes ainsi que du nom de la règle de grammaire associée au symbole de tête. Le terme `a.b.nil` se code alors de la manière suivante :

```
FSYM(FSYM(nil, ⟨a⟩).FSYM(FSYM(nil, ⟨b⟩).FSYM(nil, ⟨nil⟩).nil, ⟨@.@⟩).nil, ⟨@.@⟩)
```

3.2 Création d'outils

L'utilisation d'un environnement de programmation doit être un moyen d'améliorer la vitesse ou la qualité du cycle de développement d'un outil. Il ne doit, en aucun cas, être perçu comme un « boulet » qui engage et condamne les utilisateurs à effectuer tous leurs développements futurs dans ce même environnement. Cette idée fut l'une des principales motivations qui nous ont amenés à développer le compilateur pour ELAN.

L'utilisation du compilateur ELAN est telle qu'elle rend l'utilisation des spécifications compilées indépendantes de l'environnement de spécification : il n'est plus nécessaire de disposer du parseur, de l'interpréteur et de la bibliothèque ELAN pour exécuter une spécification. Après compilation d'une spécification, celle-ci peut être vue comme une « boîte noire » qui prend en entrée une requête et retourne des résultats conformes à ce qui a été spécifié. Ce nouveau composant peut alors être intégré dans un projet plus vaste. Nous envisageons d'ailleurs d'écrire certains composants de l'environnement ELAN en ELAN lui-même et d'utiliser le compilateur pour en faire des outils efficaces et indépendants.

Pour être utilisables, les outils générés par le compilateur doivent être capables de communiquer avec le monde extérieur. Deux solutions ont été retenues :

– Étant donné que les programmes réalisés dans l'environnement ELAN ont pour vocation de manipuler des termes, il nous a semblé naturel d'utiliser le format REF comme format d'échange entre l'extérieur et les programmes générés par le compilateur ELAN. De ce fait, toute spécification ELAN, une fois compilée, lit les requêtes dans le format REF et retourne les résultats dans ce même format. Il devient ainsi facile de réaliser et faire communiquer différents composants spécifiés en ELAN.

– Le format REF a l'avantage d'être un standard interne à l'environnement, mais il a l'inconvénient d'être difficilement lisible par un être humain. Cela a pour conséquence de rendre difficile l'écriture des requêtes et la lecture des résultats d'un programme compilé. Nous avons donc fait en sorte que tout programme compilé puisse continuer à communiquer en utilisant la syntaxe définie par l'utilisateur dans la spécification elle-même.

Ce deuxième point, d'apparence mineure, s'est révélé être un sujet d'étude intéressant. Afin de percevoir les problèmes rencontrés, il faut avoir à l'esprit la structure d'une spécification ELAN : la syntaxe des opérateurs utilisés pour définir des règles est elle-même définie dans la première partie de la spécification (la signature). Pour communiquer avec l'extérieur, le programme doit donc être capable de lire et d'écrire des termes dans cette syntaxe. D'une manière ou d'une autre, les analyseurs lexical et syntaxique doivent donc être intégrés dans le programme généré. Une solution consisterait à intégrer un générateur de parseur dans le compilateur ELAN. C'est la solution qui a été adoptée dans le projet ASF+SDF pour répondre au même type de problème. Dans ELAN, nous avons choisi l'alternative qui consiste à utiliser un algorithme général pour analyser les grammaires hors contexte (Earley 1970). L'implantation de cet algorithme d'Earley peut donc être ajoutée au code généré par le compilateur, mais pour des raisons de modularité et de réutilisation, nous avons préféré définir un outil indépendant appelé `query2ref`. Celui-ci prend en entrée deux arguments : la grammaire de la spécification (au format REF) et un terme au format défini dans la spécification ELAN, puis retourne la représentation REF de ce terme. Le codage REF peut alors être envoyé au programme compilé pour y être évalué (voir figure 3.2). L'outil inverse `ref2result` a lui aussi été défini. Son rôle consiste à lire une grammaire et un terme au format REF pour le traduire dans la syntaxe définie dans la spécification ELAN. Pour être indépendant, le programme compilé ne doit alors contenir que le codage de sa grammaire au format REF et être capable de le communiquer à ces deux outils de conversion.

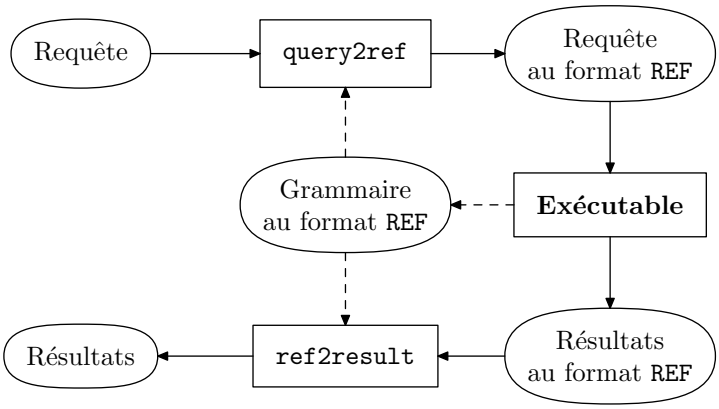


FIG. 3.2 – *Ce schéma illustre la manière dont sont organisées les entrées/sorties d'un exécutable généré par le compilateur ELAN : l'exécutable communique en utilisant le format REF et fait appel à deux utilitaires `query2ref` et `ref2result` pour effectuer les conversions en provenance et vers un format lisible. Pour fonctionner, ces deux outils ont besoin de connaître le codage au format REF de la signature de la spécification compilée. Cette signature, intégrée à l'exécutable, peut être exportée lorsque cela est nécessaire.*

3.3 Système ouvert

Disposer d'un format d'échange est un moyen de modulariser la structure interne de son environnement, mais c'est aussi un moyen de s'ouvrir vers l'extérieur. Lorsqu'on compare les différents langages de programmation issus de la communauté réécriture, il est frappant de constater que le nombre de points communs à tous ces langages est relativement important. De nombreuses tentatives ont été faites pour essayer de définir une machine abstraite pour les langages à base de règles de réécriture (Strandh 1988, Strandh 1989, Sherman 1994, Hamel 1995, Metzemakers et Sherman 1995, Kamperman 1996, Ogata et al. 1997, Ishisone et Sawada 1998). Cependant, aucune d'entre elles n'est devenue un standard comme l'est la Machine Abstraite de Warren (Warren 1983, Aït-Kaci 1990) pour la compilation de Prolog, sûrement parce que les performances de ces différentes machines abstraites n'étaient pas à la hauteur des attentes.

La démarche du projet ELAN est un peu différente, dans la mesure ou nous n'avons jamais eu l'ambition de définir une machine abstraite. Cependant, l'architecture choisie et les possibilités offertes par le compilateur nous ont amenés à tenter d'utiliser notre compilateur pour compiler d'autres langages de spécification. Dans ce cadre, une coopération entre le projet ASF+SDF d'Amsterdam et le projet ELAN de Nancy a débuté en 1998. L'objectif consistait à réaliser un outil capable de traduire une spécification ASF dans le format REF, pour pouvoir la compiler en utilisant le compilateur ELAN. Afin d'acquérir une meilleure connaissance du formalisme de spécification à traduire, cet outil a été écrit en ASF+SDF lui-même. La traduction d'un formalisme à l'autre peut se décomposer en plusieurs étapes de transformation :

- la spécification ASF est dans un premier temps traduite dans un format intermédiaire fondé sur une structure de termes : le format asFix. Au cours de cette étape, toutes les constructions infixées sont remplacées par des constructions préfixées équivalentes de sorte que le format asFix soit facile à analyser ;
- le format asFix est ensuite traduit dans un autre format intermédiaire moins riche qui ne contient plus aucune information concernant l'affichage des termes manipulés. Dans ce format appelé μASF, les noms d'opérateurs sont aussi simplifiés afin de rendre les transformations ultérieures plus faciles à effectuer ;
- différentes transformations sont appliquées sur le programme μASF pour en enlever des opérations complexes de *list-matching* : ces opérateurs associatifs sont remplacés par une nouvelle famille d'opérateurs et de règles qui simulent la réécriture modulo l'associativité ;

- seulement après ces trois étapes, le programme μASF simplifié (ne contenant plus d'opérateur associatif) peut être traduit dans le format REF : il reste à renommer les opérateurs et les constructions propres au format μASF;
- le programme REF obtenu peut alors être compilé par le compilateur ELAN pour produire un exécutable indépendant.

Ce schéma de compilation relativement complexe a été implanté avec succès et donne des résultats expérimentaux intéressants : lorsqu'on considère deux spécifications équivalentes, l'une écrite en ASF+SDF et l'autre écrite en ELAN, les deux exécutables obtenus après voir compilé leur représentation REF ont approximativement la même efficacité. Cela signifie qu'aucune surcharge n'a été introduite par les étapes successives de transformation d'ASF vers asFix, puis d'asFix vers μASF et enfin de μASF vers REF.

3.4 Vers une nouvelle architecture

Comme nous l'avons vu précédemment, bâtir l'organisation d'un environnement de spécification autour d'un format intermédiaire unique a de nombreux avantages (Borovanský, Jamoussi, Moreau et Ringeissen 1998). Au cours de différentes expériences développées dans l'équipe, le format REF s'est montré bien adapté au développement d'outils de transformation de programmes, tels qu'un évaluateur partiel permettant d'optimiser l'application des stratégies, ou qu'un débogueur ELAN écrit en ELAN. Le format REF a cependant le défaut d'être trop proche de la représentation interne des données du parseur et de l'interpréteur. Cette proximité nous a permis de développer et d'expérimenter rapidement les possibilités offertes par cette nouvelle architecture, mais avec le temps, certaines limitations se font sentir. Il est en particulier impossible de coder dans le format REF les noms des variables utilisées dans les spécifications originelles. Cela vient du fait que les noms des variables sont « perdus » lors des phases d'analyse lexicale et syntaxique du parseur ELAN.

Tout en conservant l'idée d'avoir un format intermédiaire, cela nous a amené à définir une nouvelle structure plus modulaire de ce format d'échange. Profitant de l'expérience du groupe ASF+SDF et de la venue à Nancy de Mark van den Brand, nous nous sommes inspirés du format asFix pour définir le format Efix. Ces deux formats reposent sur les notions de termes annotés (ATerms) et de syntaxe abstraite du langage.

ATerms est un formalisme générique qui permet de représenter des informations structurées telles que des arbres syntaxiques. L'un des principaux intérêts de ce formalisme est d'être lisible par un humain et facilement manipulable par un ordinateur. La syntaxe concrète d'ATerms se présente en ELAN de la manière suivante :

```
module aterm
import global int string;
end
sort ATerms ATermList AFun ATerm Ann;
end
operators global
  @               : (ATerm)           ATerms;
  @ , @           : (ATerm ATerms)    ATerms;
  []              :                   ATermList;
  [ @ ]           : (ATerms)          ATermList;
  @               : (int)             AFun;
  @               : (string)          AFun;
  @               : (ATermList)       ATerm;
  @               : (AFun)            ATerm;
  @ ( @ )         : (AFun ATerms)     ATerm;
  < @ >           : (ATerm)           ATerm;
  '\123' @ '\125' : (ATerms)          Ann;          // Définition de '{' @ '}'
  @ @             : (ATermList Ann)   ATerm;
  @ @             : (AFun Ann)        ATerm;
  @ ( @ ) @       : (AFun ATerms Ann) ATerm;
  < @ > @         : (ATerm Ann)       ATerm;
end // operators
end // module
```

C'est en instanciant la sorte `AFun` que des versions spécifiques de ce formalisme peuvent

être créées. La version définie pour ASF+SDF s'appelle asFix et la version pour ELAN s'appelle Efix. À la différence du format REF, ce nouveau format est complètement fondé sur la syntaxe abstraite du langage ELAN. Il devient alors indépendant de toute implantation, et en particulier, des structures de données du parseur.

Considérons par exemple la façon dont un module ELAN est représenté dans ce nouveau format. Partant de la syntaxe abstraite :

```
<Module> ::= module ( <FormalModuleName>,
                      <Imports>,
                      <SortDefinition>,
                      <OperatorDefinition>,
                      <StrategyDefinition>,
                      [{<FamilyOfRule> ","}*],
                      [{<FamilyOfStrategies> ","}*] )
```

Il faut noter que les crochets (`[` et `]`) ne définissent pas des paramètres optionnels mais des listes de paramètres comme décrit par le formalisme ATerms. L'implantation ELAN correspondante est la suivante :

```
operators global
  module @ @ @ @ @ @ @ End :
    (FormalModuleName ImportsOpt SortDefinitionOpt
     OperatorDefinitionOpt StrategyDefinitionOpt
     ListOfFamilyOfRules ListOfFamilyOfStrategies) Module;
```

La sorte `Module` doit ici être vue comme une instanciation pour ELAN de la sorte `AFun` définie dans les ATerms.

Bien que moins compact que le format REF, un des intérêt de ce nouveau format est d'être lisible mais surtout modulaire : une expression du format Efix peut aussi bien correspondre à un simple terme tel que `a.b.nil`, à un module de spécification ou à une spécification toute entière. Cet aspect devient important lorsqu'il s'agit de réaliser un parseur ou un préprocesseur pour ELAN qui soit capable d'analyser des extraits de spécifications. Dans le cadre d'un langage compilé, il est aussi intéressant de pouvoir représenter chaque module par un terme différent, afin d'offrir des possibilités de compilation modulaire et séparée.

Même si l'implantation actuelle de l'environnement ELAN est fondée sur l'usage d'un langage de commande de type *shell* et que les spécifications sont écrites en utilisant un éditeur de texte du type *emacs*, dans l'optique de définir de nouvelles fonctionnalités de l'environnement, il est nécessaire d'établir des *scenarii* d'actions d'utilisateurs telles que celles présentées ci-dessous :

- l'utilisateur veut éditer des modules ;
- l'utilisateur veut utiliser un module pour évaluer une requête en utilisant l'interpréteur, sachant que le module utilisé n'est pas toujours le même ;
- l'utilisateur veut mettre en page et *pretty-printer* un module ;
- l'utilisateur veut compiler une spécification et utiliser le code généré pour évaluer une requête ;
- l'utilisateur veut (interactivement) déboguer une spécification.

À partir de ces *scenarii*, nous pouvons imaginer quels sont les composants et les fonctionnalités qu'il doit être possible d'intégrer dans le nouvel environnement. Un exemple d'environnement ELAN est présenté dans la figure 3.3.

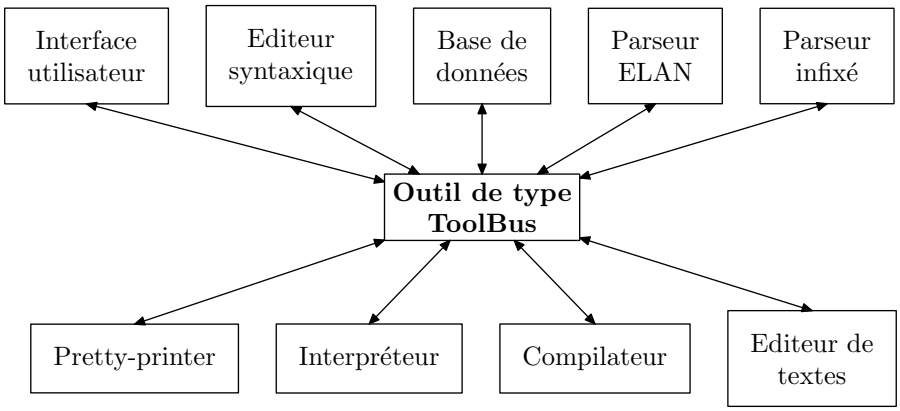


FIG. 3.3 – *Vers une nouvelle architecture de l'environnement ELAN.*

Base de données de modules. C'est un composant essentiel de l'environnement qui doit offrir un mécanisme flexible pour parcourir une relation d'importation et retrouver les modules correspondants. Cette base de données joue un rôle très important, parce que c'est elle qui mémorise quels sont les modules qui doivent être analysés ou compilés, par exemple, pour être capable de réécrire un terme. Elle doit aussi gérer le statut des modules pour savoir, lors d'une recherche, quels sont les modules qui doivent être recherchés sur le disque et ceux qui ont été modifiés par des actions d'édition.

Éditeur de texte. L'éditeur doit permettre d'éditer des fichiers textuels, mais aussi être capable de se connecter à un outil extérieur tel que le ToolBus pour lui communiquer le contenu de ses fichiers.

Éditeur syntaxique. La distinction entre un éditeur de texte et un éditeur syntaxique peut sembler artificielle, mais ces deux outils fournissent des services complètement différents : l'éditeur de texte permet d'insérer ou de supprimer des caractères alors qu'un éditeur syntaxique permet de manipuler la structure syntaxique du texte. Lorsque qu'un programme, saisi avec l'éditeur de texte, devient (après analyse) syntaxiquement correct, l'éditeur syntaxique peut être utilisé pour offrir des fonctionnalités telles que la sélection, l'effacement ou la modification de sous-arbres.

Parseur ELAN et Parseur de termes infixés. Une des idées de cette nouvelle architecture pour ELAN est de séparer clairement les étapes d'analyse, de pré-traitement et d'interprétation brièvement décrites dans le paragraphe 2.2. Le format Efix semble être un bon point de départ qui nous permettrait de représenter un module ELAN au cours de son analyse syntaxique :

- partant d'un module ELAN, les parties analysables par un outil comparable à *Yacc* sont, dans un premier temps, lues et transformées dans leur représentation Efix. Les morceaux de texte correspondant à des termes infixés ou à des constructions du préprocesseur sont alors mémorisés par une suite de caractères (non analysée) dans une représentation Efix.
- partant de cette représentation Efix intermédiaire (parce que non complètement analysée), la grammaire de la signature du module peut être lue et utilisée pour construire un analyseur de termes infixés (l'algorithme d'Earley ou des techniques de génération de parseurs peuvent être utilisées ici). Disposant d'un analyseur de termes infixés, les morceaux de

textes du fichier Efix intermédiaire qui correspondent à des termes non analysés, peuvent alors être lus et remplacés par leur codage Efix correspondant.

– la dernière étape consiste à analyser, puis évaluer les constructions du préprocesseur mémorisées dans la représentation Efix intermédiaire d'un module. Il faut pour cela utiliser les morceaux analysés afin de construire un système de calcul à l'aide de l'interpréteur ou du compilateur. Ce système de calcul peut alors servir à évaluer les constructions du préprocesseur pour les remplacer par leur codage Efix correspondant.

Partant d'une spécification ELAN, les trois étapes précédentes sont appliquées itérativement, jusqu'à obtenir un point fixe qui correspond à une représentation Efix de la spécification ne contenant plus de morceau de texte non analysé ou non évalué par le préprocesseur. En suivant cette approche il devient possible de séparer complètement les phases d'analyse syntaxique de la partie « fixe » d'ELAN, l'analyse des termes infixés dépendant d'une grammaire hors contexte, et les phases d'évaluation du préprocesseur.

Pretty-printer. Cet outil s'occupe de mettre en page et de rendre lisible les termes. Il peut, par exemple, être utilisé pour imprimer des modules ELAN au format HTML ou LaTeX.

Interpréteur et Compilateur. Ces outils correspondent à ce qui a été présenté dans le chapitre 2.

Interface utilisateur. Cette interface aide à visualiser l'ensemble des différents composants définis précédemment, ou l'ensemble des modules qui composent une spécification par exemple.

3.5 Synthèse

Dans ce chapitre, nous avons présenté l'architecture générale de l'environnement de spécification ELAN, et plus précisemment son organisation autour du format d'échange REF. Ce format d'échange permet d'une part de connecter et d'intégrer le nouveau compilateur dans l'environnement ELAN ou ASF+SDF, mais il est aussi source d'ouverture en facilitant l'inter-connexion des outils générés par le compilateur lui-même. Comme le précise le paragraphe 3.4, la définition d'un format d'échange suffisemment modulaire, lisible et général est un thème de recherche encore d'actualité. En collaboration et s'appuyant sur l'expérience du groupe ASF+SDF, nous proposons le format Efix fondé sur la notion de termes annotés. Ce nouveau format d'échange nous permettra à terme de modulariser l'environnement ELAN, ce qui facilitera la rénovation, le développement et l'intégration de nouveaux composants. Il devrait aussi faciliter l'interaction et la réutilisation des outils développés par les groupes ASF+SDF, CafeOBJ, Maude et ELAN par exemple. À l'image d'autres communautés, un même langage pourrait ainsi disposer de plusieurs implantations et inversement, un même outil pourrait être utilisé pour compiler différents langages.

Deuxième partie

Compilation de la réécriture

Chapitre 4

Méta-conception

4.1 Interpréteur, Compilateur et Machine abstraite 53
4.2 Pourquoi choisir un compilateur 56
4.3 Compilation de la réécriture 58

Plusieurs niveaux de conception interviennent dans le cycle de vie d'un logiciel. Nous avons déjà abordé dans le chapitre 3 les liens existant entre la conception de l'environnement de spécification et celle d'un composant. Étant donné un composant, le compilateur par exemple, il faut distinguer sa conception interne d'une part, qui décrit son fonctionnement et les interactions entre ses différentes phases de transformation ; et sa conception d'un point de vue plus général, appelée « méta-conception ». C'est à ce niveau d'étude que des choix importants sont effectués. Même s'il est toujours préférable de réaliser les outils les plus modulaires et les moins figés possible, il est dans certains cas plus profitable de faire des choix de base qui vont influencer les solutions techniques à mettre en œuvre.

Dans l'industrie automobile par exemple, lorsqu'un nouveau modèle de voiture est conçu, il est nécessaire de faire des « méta-choix » avant de concevoir des solutions à mettre en place. Ainsi, avant d'étudier les problèmes de transmission par exemple, il est préférable de connaître le nombre de roues de la voiture et de savoir s'il s'agit d'une traction avant ou d'un modèle à propulsion. La remise en cause de ces choix n'est pas toujours aisée : partir d'une voiture à trois roues et lui ajouter une roue n'est peut être pas le meilleur moyen pour obtenir une « bonne » voiture à quatre roues.

La recherche ou l'industrie informatique travaille avec des matériaux un peu moins tangibles, mais cela ne nous dispense pas des étapes de conception, bien au contraire. L'objectif de ce chapitre est de « dessiner » les grandes lignes du système que nous voulons construire : un compilateur pour ELAN capable de produire des outils suffisamment efficaces.

4.1 Interpréteur, Compilateur et Machine abstraite

Pour qu'un programme écrit dans un certain langage L puisse être exécuté sur un ordinateur, on doit rendre disponible ce langage sur cet ordinateur. D'une manière générale, il existe deux façons d'*implanter* un langage sur un calculateur : réaliser un interpréteur ou un compilateur pour ce langage.

Un interpréteur I_L pour un langage L est un outil qui prend en entrée un programme p_L écrit dans le langage L et la suite des données d'entrée e du programme p_L ; l'interpréteur calcule la

suite des résultats r. L'interprétation d'un programme peut éventuellement mener à une erreur, c'est pourquoi la signature fonctionnelle d'un interpréteur est la suivante :

$$I_L : L \times D^* \mapsto D^* \cup \{erreur\},$$

où D est le domaine des données d'entrée et de sortie du programme p_L. D'un point de vue formel, l'interprétation d'un programme p_L est spécifiée par l'équation suivante :

$$I_L(p_L,e) = r$$

Ce qui caractérise un interpréteur, c'est qu'il travaille simultanément sur le programme p_L et ses données e. D'une manière générale, un interpréteur n'essaie pas d'analyser le contenu d'un programme avant de l'exécuter : les instructions du programme p_L sont *décodées* et *interprétées* les unes à la suite des autres (voir figure 4.1).

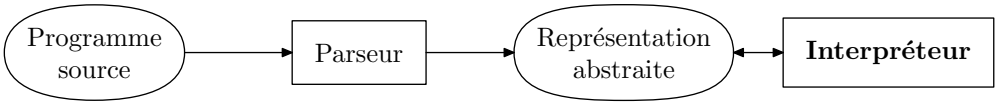


FIG. 4.1 – *Dans le cadre d'un interpréteur, le programme est lu par le parseur, puis les instructions sont décodées et interprétées les unes à la suite des autres par l'interpréteur.*

Il existe cependant des techniques de « semi-compilation » qui permettent de réaliser des interpréteurs qui effectuent une analyse préalable du programme à interpréter. Les informations *statiques* peuvent ainsi être utilisées pour améliorer la vitesse d'exécution du programme p_L (voir figure 4.2).

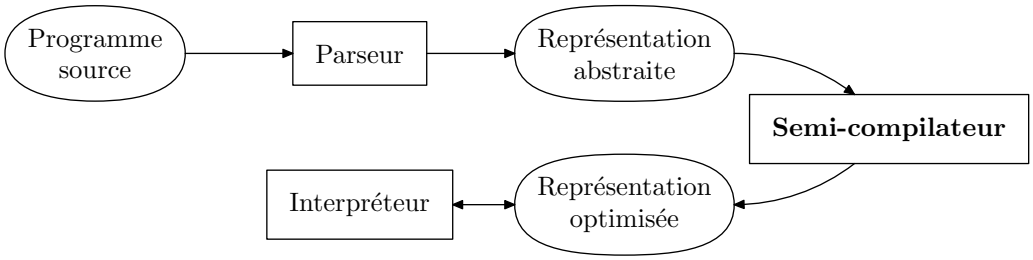


FIG. 4.2 – *Dans le cadre d'un semi-compilateur, le programme est lu par le parseur, puis la représentation abstraite du programme est transformée (compilée) pour obtenir une nouvelle représentation abstraite optimisée qui sera exécutée en utilisant un interpréteur.*

Contrairement à un interpréteur, un compilateur pour un langage L ne permet pas d'exécuter directement un programme p_L écrit dans le langage L. Un compilateur n'est rien d'autre qu'un outil permettant de traduire un programme écrit dans un langage L, appelé langage source, vers un autre programme *équivalent* écrit dans un autre langage M, appelé langage cible (voir figure 4.3). Deux programmes p_L et p_M sont *équivalents* si à partir des même données e ils retournent les mêmes résultats r. En supposant qu'il existe deux interpréteurs I_L et I_M, on aurait :

$$p_L \text{ équivalent à } p_M \iff I_L(p_L,e) = I_M(p_M,e) = r$$

La situation semble assez claire : les compilateurs servent à traduire des programmes d'un langage vers un autre et les interpréteurs permettent de les exécuter.

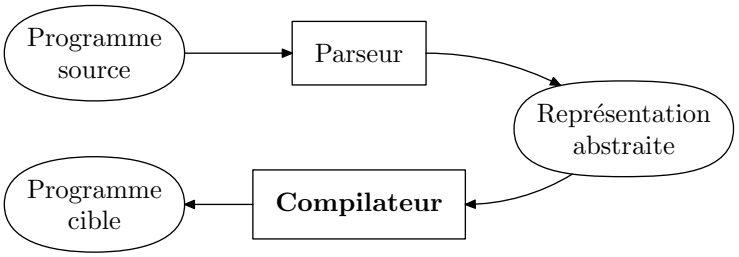


FIG. 4.3 – *Dans le cadre d'un compilateur, le programme source est lu par le parseur. À la différence d'un interpréteur, la représentation abstraite n'est pas exécutée mais traduite dans un nouveau langage (appelé cible). Ce langage cible peut à nouveau être compilé ou interprété.*

Pourquoi dire qu'il existe deux façons d'implanter un langage sur un calculateur : réaliser un interpréteur ou un compilateur, si seuls les interpréteurs permettent d'exécuter les programmes ? Pour sortir de ce « paradoxe », le calculateur doit avoir une certaine propriété : il doit être capable d'exécuter un langage donné, appelé *langage machine*. Cette propriété est tellement sous-entendue, qu'on peut considérer comme axiome de base le fait qu'un ordinateur possède une unité centrale, appelée *processeur*, capable de comprendre au moins un langage : le langage machine. On comprend mieux maintenant comment un compilateur permet de rendre exécutable un programme écrit dans le langage L : il suffit de le traduire en un programme écrit en langage machine. Celui-ci peut alors être directement exécuté par le processeur. Pour le non spécialiste en électronique, un processeur est un ensemble de composants électroniques qui permettent d'effectuer des opérations élémentaires telles qu'une addition ou la mise en mémoire d'une information par exemple. Il est donc assez réaliste de voir le processeur comme une « boite noire» capable de comprendre directement le langage machine. Et pourtant la réalité est souvent un peu plus complexe. Aussi étrange que cela puisse paraître, le langage machine n'est pas assez « primitif» pour que l'on puisse réaliser facilement des circuits électroniques capables d'exécuter les opérations du langage. Les fabricants ont donc ajouté un niveau intermédiaire appelé *micro code*. Cette fois-ci, il existe bien des assemblages de composants électroniques qui permettent d'exécuter directement des programmes écrits dans ce langage. Les fabricants réalisent alors un interpréteur, écrit en micro-code, pour le langage machine qu'ils veulent mettre à disposition des utilisateurs. D'un point de vue extérieur, peu importe le nombre de niveaux intermédiaires ajoutés, le processeur se comporte comme s'il comprenait directement le langage machine. Mais d'un point de vue conceptuel, l'existence d'un interpréteur est toujours nécessaire pour exécuter un programme. C'est pourquoi il n'est jamais évident d'affirmer que tel programme est interprété et tel autre compilé. Tout dépend du niveau de granularité avec lequel le processeur est considéré.

Supposons que nous voulions « implanter » un langage L sur un ordinateur, vaut-il mieux créer un interpréteur ou un compilateur pour le langage L ? Il n'existe évidement pas de réponse unique. Le choix dépend de ce qu'on attend de l'implantation du langage L. Veut-on un outil rapide à réaliser ? adaptable à des extensions éventuelles du langage L ? réutilisable pour implanter un autre langage L' ? fonctionnant sur différents types de processeurs ? permettant d'exécuter rapidement un programme p_L ? autorisant des modifications dynamiques d'un programme p_L ? offrant un « cycle de développement » relativement court ? permettant d'intégrer ou de faire coopérer un programme p_L avec un autre programme $p_{L'}$?

Le rêve de tout informaticien est de réaliser un produit répondant par l'affirmative à toutes

ces questions, mais l'état de l'art actuel ne permet de répondre que partiellement à l'ensemble de ces questions. Le choix doit donc se faire en fonction de priorités pré-définies. On a souvent considéré qu'un interpréteur permettait d'implanter plus facilement un langage de « haut niveau » et qu'un compilateur rendait plus efficace l'exécution des programmes. Cette vision simpliste a du vrai, mais il ne faut pas oublier le développement de techniques hybrides qui permettent de mélanger des phases de compilation avec des phases d'interprétation. On parle alors de « semi-compilation » ou de « machines abstraites ». Par opposition aux processeurs qui sont des outils tangibles permettant d'exécuter le langage machine, les outils logiciels qui permettent d'exécuter un langage L sont appelés des « machines abstraites », et ceci, indépendemment de la façon dont ils sont implantés. Pour implanter un langage L, on peut donc choisir de réaliser une machine abstraite pour un langage L' à l'aide d'un interpréteur, puis de réaliser un compilateur de L vers L'. La vitesse d'exécution de l'ensemble dépendra en partie du niveau choisi pour le langage L' et de la qualité de son interpréteur. Cette approche a l'avantage d'être « incrémentale » : définir et implanter une machine abstraite avec un interpréteur peut être une solution facile et rapide à mettre en œuvre, et rien n'empêche de remplacer l'interpréteur par un compilateur si les performances ne sont pas assez bonnes. Cette stratégie a été largement suivie dans de nombreuses implantations de langages fonctionnels ou logiques.

La définition d'un jeu d'instructions pour Prolog (Warren 1983), appelé WAM (Warren Abstract Machine (Aït-Kaci 1990)) par la suite, fut à l'origine de progrès majeurs concernant l'implantation de Prolog. De nombreux interpréteurs ont été implantés pour ce langage abstrait, et depuis peu, des compilateurs tels que Wamcc (Codognet et Diaz 1995, Diaz 1995) ont été réalisés pour offrir des implantations plus efficaces du langage.

4.2 Pourquoi choisir un compilateur

Le projet ELAN a réellement démarré en 1991 lorsque Marian Vittek a entamé la conception et la réalisation des outils adaptés au langage ELAN. La simplicité et la bonne compréhension des mécanismes ont toujours guidé les choix effectués au cours de la conception et de la réalisation.

La recherche d'idées simples n'a cependant pas empêché l'émergence de deux idées nouvelles qui différencient ELAN de tous les autres systèmes fondés sur la logique de réécriture : l'intégration au parseur d'un préprocesseur puissant et l'existence d'un langage spécifique pour décrire des stratégies et offrir un meilleur contrôle sur l'application des règles de réécriture.

Malgré quelques petits défauts, l'environnement s'est montré agréable à utiliser et a rapidement mené les utilisateurs à écrire des spécifications relativement longues et complexes. La taille des termes manipulés et la taille de l'espace de recherche lié à l'exploration des stratégies sont devenues vraiment grandes, le nombre de règles et de stratégies relativement important, quant au nombre moyen d'étapes de réécriture nécessaires pour mener à bien un calcul, il a lui aussi augmenté de manière significative. Cette première implantation a montré l'intérêt pratique des théories et techniques de réécriture développées dans ce domaine depuis plusieurs années. Mais il a aussi montré les difficultés pour trouver ce compromis entre l'expressivité et l'efficacité qui incite les chercheurs à implanter leurs outils en utilisant un langage fondé sur la réécriture.

Étant convaincu de la qualité des langages de spécification fondés sur la réécriture, nous avons décidé de porter nos efforts sur l'élaboration de méthodes nous permettant de réaliser un environnement de spécification utilisable pour des applications « grandeur nature ». Notre objectif consiste donc à mettre en place un support d'exécution pour le langage ELAN qui soit capable de manipuler des termes et des spécifications de grande taille, tout en garantissant une certaine rapidité d'exécution des spécifications écrites en ELAN. Pour atteindre cet objectif,

l'alternative était de réaliser soit un très bon interpréteur, soit un bon compilateur.

La première alternative est une solution ambitieuse qui demande une précision, une rigueur et des qualités de programmeur exemplaires pour pouvoir se démarquer de l'ensemble des interpréteurs existants.

La compilation de la réécriture n'est pas non plus un domaine récent (Hoffmann et O'Donnell 1982b). De nombreuses tentatives ont été faites pour essayer de réaliser des compilateurs de systèmes de réécriture (Strandh 1988, Sherman 1994, Hamel 1995, Metzemakers et Sherman 1995, Kamperman 1996, Ogata et al. 1997, Ishisone et Sawada 1998), et pourtant, l'histoire montre qu'aucun ne s'est imposé. Peut être parce qu'une grande majorité des tentatives ont suivi la même approche : définir, créer et utiliser une machine abstraite pour compiler la réécriture. Il est clair que la compilation des langages logiques ou fonctionnels est un domaine connexe, mais les solutions à mettre en œuvre ne sont pas tout à fait du même ordre.

La nouveauté ou l'inconnu de notre approche est de tenter de se passer d'une machine abstraite et de considérer qu'un langage impératif tel que le `C` est finalement bien adapté à la compilation des systèmes de réécriture. Le pari a débuté en 1995 lorsque Marian Vittek a entamé l'écriture d'un premier compilateur pour ELAN. Un an plus tard, le compilateur commençait à donner ses premiers résultats : les performances pouvaient être qualifiées d'« extra-ordinaires » à l'époque. Ces premiers résultats ont eu une grande importance car ils ont eu pour effet de convaincre une partie de la communauté scientifique qu'un langage de programmation fondée sur la réécriture n'est pas condamné à rester isolé sur une machine d'un centre de recherche. Malheureusement, certains choix effectués pour réaliser ce prototype n'ont pas permis son extension aux évolutions ultérieures du langage de spécification lui-même. Le prototype avait rempli son rôle, et un nouveau développement intégrant dès sa conception des objectifs à plus long terme s'est avéré nécessaire.

L'idée de réaliser un nouvel outil efficace, robuste et modifiable a motivé particulièrement cette thèse.

D'un point de vue pratique, le compilateur doit être robuste : il doit d'une part être capable de compiler des spécifications de grande taille, mais il doit aussi générer du code de bonne qualité, capable d'effectuer des calculs pouvant durer plusieurs jours sans faire d'erreur ou consommer trop de mémoire. Le code généré doit évidemment être suffisamment efficace, mais ce n'est pas une priorité absolue, ce qui signifie qu'en cas d'hésitation ou de doute, la clarté et la qualité du compilateur doit être prépondérante sur la vitesse d'exécution du code produit.

D'un point de vue scientifique, le développement de ce projet doit permettre d'harmoniser et de faire cohabiter la plupart des techniques existantes, mais il doit aussi permettre l'innovation et le développement de nouveaux algorithmes. Le projet ne se limite donc pas à refaire en mieux le prototype réalisé par Marian Vittek : le compilateur doit aussi permettre d'exécuter la réécriture modulo l'Associativité et la Commutativité. Ces deux mots peuvent paraître anodins et pourtant l'organisation du nouveau compilateur doit être complètement modifiée pour pouvoir gérer des systèmes de réécriture modulo une certaine théorie E. Cela signifie entre autres que la structure des termes manipulés, la compilation du filtrage, la gestion des retours arrière et la gestion mémoire doivent être totalement repensées.

Une fois établies ces grandes lignes directrices, il reste encore à faire deux choix principaux : dans quel langage écrire le compilateur ? et quel type de langage cible utiliser ? Faut-il continuer à générer du `C` ou faut-il revenir à une méthode plus classique fondée sur l'utilisation d'une machine abstraite ?

Pour les raisons présentées précédemment, le deuxième choix s'est fait assez naturellement et la volonté de générer du `C` est maintenue. Quant à la première question, la réponse n'est vraiment pas évidente. Il faut choisir un langage bien adapté à la réalisation des compilateurs,

qui permette le développement en équipe et qui respecte les critères énoncés précédemment. Il faut aussi que ce langage soit suffisamment enseigné dans le milieu universitaire pour que les étudiants amenés à travailler sur le projet puissent s'intégrer sans trop de difficulté. Nous avons étudié quatre possibilités qui ont chacune leurs avantages et leurs inconvénients :

- utiliser un langage impératif tel que le C ou le C++ : c'est une solution sûre qui a l'avantage d'uniformiser les langages de développement utilisés. L'interpréteur étant implanté en C++, une partie des bibliothèques peut être facilement réutilisée. Mais même si les étudiants ont en général une bonne connaissance de ces langages, leur souplesse peut être responsable de très nombreuses « petites » erreurs si une méthode rigoureuse empruntée au monde industriel n'est pas appliquée.
- utiliser un langage fonctionnel du type Caml : c'est sûrement une très bonne solution, mais c'est un langage encore relativement peu utilisé en dehors du territoire français. Ce n'est pas gênant en soi, mais cela peut devenir un handicap dans le cadre d'un développement réparti entre plusieurs équipes internationales par exemple.
- utiliser ELAN lui-même : les techniques d'« amorçage » ou de *bootstrapping* ont souvent un impact bénéfique considérable sur la qualité du langage développé. Cela permet en effet de tester en permanence la plupart des constructions du langage et les qualités du compilateur développé. Il faut cependant disposer d'un langage et d'outils suffisamment figés pour ne pas mener tous les combats en même temps. ELAN est un produit de recherche en constante évolution et bien que le langage et les outils s'améliorent jour après jour, le risque est dans ce cas de devoir développer et maintenir en parallèle le langage, l'interpréteur et le compilateur.
- utiliser un langage à objets tel Eiffel ou Java : ces langages permettent en général d'améliorer considérablement la qualité du code développé et d'augmenter leur réutilisabilité. Choisir entre Eiffel et Java n'est pas simple. Eiffel a l'avantage d'être relativement puissant, stable et efficace. De plus, la présence dans notre laboratoire des auteurs du premier compilateur GNU Eiffel n'aurait dû laisser aucune chance au nouveau langage qu'était Java en 1996. Et pourtant, c'est le choix inverse qui a été fait. L'effet de mode de l'époque a sûrement eu une influence non négligeable sur cette décision, mais c'est aussi l'impression de fiabilité dégagée par la lecture des spécifications du langage et quelques expérimentations des outils qui nous ont amenés à tenter l'aventure. Il faut avouer que c'était un pari un peu risqué à l'époque parce qu'il n'y avait pas l'engouement que l'on connaît aujourd'hui. Mais l'avenir nous a donné raison et si le choix était à refaire, ce serait sans la moindre hésitation que nous choisirions de nouveau Java pour être le langage d'implantation du compilateur. Mis à part le fait d'être portable et orienté objets, Java est un langage vraiment agréable à utiliser tous les jours pour réaliser ce type d'application. Les outils de développement et la documentation sont bien pensés et la qualité des mécanismes de gestion d'erreurs sont d'une aide inqualifiable.

Les « méta-choix » étant faits (choix des objectifs prioritaires, choix du langage cible et choix du langage d'implantation), il reste à définir les grandes lignes du compilateur (représentation des règles et des termes) avant d'expliquer en détail comment celui-ci a été réalisé.

4.3 Compilation de la réécriture

On peut avoir deux approches pour compiler un langage donné. La première consiste à représenter les structures dominantes du langage source par des structures de *données* du langage cible qui sont ensuite évaluées par un ensemble de fonctions bien définies. Cette approche permet

bien de créer un exécutable indépendant à partir d'un programme initial, mais elle est basée sur une certaine « tricherie ». Représenter la spécification initiale par des structures de données revient à définir, sans l'expliciter clairement, un langage intermédiaire destiné à être interprété par une machine abstraite. Et définir un ensemble de fonctions capables de donner un sens à ces structures de données n'est rien d'autre que la réalisation d'un interpréteur ou d'une machine abstraite. Ce schéma, appelé dans certains cas « compilation », s'apparente plus à une approche hybride tendant à faire cohabiter dans un même exécutable une machine abstraite et le code qu'elle doit interpréter.

La deuxième approche consiste à représenter les caractéristiques du langage source par des structures de *contrôle* du langage cible, ce qui est fondamentalement différent. Comme nous le verrons par la suite, il est dans certains cas très difficile, voir impossible, d'établir cette correspondance. Il faut alors avoir recours à la technique hybride présentée plus haut, mais tout l'art de la compilation consiste à minimiser le plus possible ces « écarts ». Considérons par exemple le prédicat `assert` de Prolog qui permet de modifier dynamiquement un programme en lui ajoutant des clauses qui ne sont pas connues au moment de la compilation. Il est alors impossible de générer uniquement des structures de contrôle du langage `C`, par exemple, qui permettent de donner un sens à la clause qui est encore inconnue. Il est d'une manière générale impossible de se passer de l'approche hybride lorsque le langage source contient des constructions ayant trait à la réflexivité.

En choisissant d'utiliser le `C` comme langage cible, nous savions que les extensions réflexives d'ELAN, telles que la création dynamique de stratégies, ne pourraient pas être compilées en utilisant le même schéma que celui conçu pour les règles et les stratégies définies statiquement. Mais ce n'est pas un handicap en soi : il parait tout à fait acceptable que les stratégies définies dynamiquement (qui représentent une infime partie des stratégies utilisées dans la pratique) s'exécutent de manière moins efficace que les autres. Et je pense que dans le cadre d'ELAN, il est nettement préférable de ne pas pénaliser l'exécution de l'ensemble du système de réécriture en utilisant une technique de compilation particulièrement dédiée.

Nous avons donc choisi de représenter une grande partie des unités syntaxiques d'ELAN, à savoir les opérateurs, les règles de réécriture et les stratégies, par des fonctions du langage `C`. Les variables et les constructeurs sont quant à eux représentés par des structures de données du `C` : il faut bien allouer des morceaux de mémoire pour représenter et mémoriser les termes.

Nous avons présenté la manière dont les structures de données du langage cible étaient traduites. Il reste à étudier comment les actions sur ces données peuvent se traduire en des actions dans le formalisme du langage cible. Que deviennent la sélection et l'application d'une règle, par exemple ?

Nous avons choisi de regrouper et de traduire en une seule fonction `C` les règles commençant par un même symbole de tête (voir figure 4.4). Cela implique qu'à chaque symbole pouvant apparaître en tête du membre gauche d'une règle, est associée une fonction. Ces symboles sont dits *définis* et les autres sont des *constructeurs*. Le fait d'utiliser une stratégie *leftmost-innermost* évite d'avoir à construire le terme avant d'essayer de le réduire, car ces deux étapes peuvent être fusionnées en une seule : les termes sont construits en partant des feuilles et chaque fois qu'un symbole constructeur apparaît, une zone de mémoire est allouée pour le représenter. Lorsqu'un symbole défini apparaît, cela signifie que le terme est potentiellement réductible puisqu'il existe au moins une règle commençant par ce symbole. La fonction associée à ce symbole est alors appelée. Son rôle consiste à déterminer si une règle peut s'appliquer et à réduire le terme courant lorsque c'est possible. La fonction `C` se compose en fait de deux parties : la première implante une procédure de filtrage qui, étant donné un terme clos, sélectionne l'ensemble des règles qui peuvent s'appliquer. La deuxième partie a pour but de sélectionner une règle parmi cet ensemble

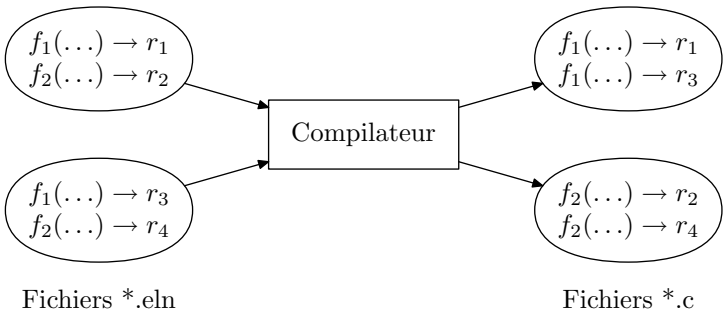


FIG. 4.4 – *Cette figure illustre l'approche consistant à regrouper les règles de réécriture commençant par un même symbole pour générer une fonction* `C` *par symbole de tête différent. Il faut remarquer que les symboles* f_1 *et* f_2*, qui étaient définis dans des fichiers différents, sont regroupés dans des fichiers* `C` *identiques.*

et d'effectuer son application : le membre droit de la règle est instancié pour construire le terme réduit. Lorsqu'aucune règle ne peut s'appliquer, c'est que le terme d'entrée est irréductible et il est retourné sans être modifié. Il faut noter que les termes retournés par les fonctions `C` sont toujours en forme normale, par construction.

L'application des règles de réécriture conditionnelles se base sur la même approche : avant de construire le terme réduit, chaque terme correspondant à une condition est construit, mis en forme normale puis comparé à la constante `true`. En cas d'égalité, l'exécution se poursuit par l'instanciation du membre droit de la règle. En cas d'inégalité, les fonctions permettant de gérer les retours arrière sont utilisées pour extraire d'autres solutions. Si, finalement, la règle courante ne peut pas s'appliquer, une autre règle de l'ensemble, engendrée par la première étape, est sélectionnée.

Il faut retenir de cette partie que notre approche est particulière, dans la mesure où des ensembles de règles sont traduits en des fonctions `C`, chaque stratégie est également représentée par une fonction `C`, et les termes ne sont jamais totalement construits en mémoire : c'est une combinaison de constructions et d'appels de fonctions qui permet d'obtenir les formes normales désirées.

Chapitre 5

Compilation du filtrage syntaxique

5.1 Termes vus comme des chaînes de symboles 62
5.2 Automate de filtrage 63
5.3 Clôtures d'un ensemble de motifs 66
5.4 Clôture « réduite » d'un ensemble de motifs 70
5.5 Automate de filtrage à mémoire 72
5.6 Automate de filtrage avec jumpNode 75
5.7 Comparaison des différentes approches 78

Dans tout système dont le mécanisme d'évaluation repose sur l'application de règles de transformations, l'étape de sélection de la règle à appliquer est importante. C'est elle qui détermine en partie la suite du calcul, et c'est aussi elle qui est exécutée le plus grand nombre de fois. Dans le cadre de la réécriture, cette sélection est faite après chaque étape de réduction, et c'est pourquoi elle doit être particulièrement étudiée pour ne pas pénaliser l'ensemble du processus.

Étant donné un terme clos, appelé sujet, le problème consiste à sélectionner une règle qui permet de le réduire. On distingue alors deux catégories de problèmes de filtrage différents, suivant que les réductions se font seulement à la racine du sujet ou à des positions correspondant à des sous-termes du sujet. Ces derniers problèmes, dits *complets* sont liés à l'étude de stratégies parallèles de réduction et sont à rapprocher des problèmes de filtrages sur les mots (Aho et Corasick 1975, Hoffmann et O'Donnell 1982a). Dans le cadre de la réécriture séquentielle, les problèmes de filtrage (*pattern-matching*) font partie de la première catégorie et on s'intéresse alors aux algorithmes qui permettent de sélectionner efficacement une règle parmi un ensemble. L'approche consistant à sélectionner une règle avant de savoir si elle peut s'appliquer, est dite *one-to-one* parce que les problèmes de filtrage ne font intervenir qu'une seule règle de réécriture (et un seul sujet) à chaque fois. On imagine facilement que ce type d'approche est peu efficace parce que sa complexité est proportionnelle au nombre de règles composant le système. C'est pourquoi des méthodes dites *many-to-one* ont été développées (Gräf 1991, Sekar, Ramesh et Ramakrishnan 1992, Graf 1996, Nedjah 1997, Nedjah, Walter et Eldrige 1997) : elles permettent de déterminer efficacement une règle du système permettant de réduire le sujet. Certaines de ces méthodes, (Gräf 1991, Nedjah et al. 1997) par exemple, sont dites *déterministes* parce qu'elles permettent de déterminer l'ensemble des règles pouvant s'appliquer sur le sujet, pour un coût comparable aux algorithmes ne sélectionnant qu'une seule règle.

Dans le cadre de la théorie syntaxique, les motifs ne sont composés que de symboles de fonctions, de constantes et de variables. Le sujet est un terme clos : il ne comporte que des constantes

ou symboles constructeurs. Le problème consiste à organiser intelligemment l'ensemble de motifs pour être capable, pour un sujet donné, de déterminer rapidement quels sont les motifs qui filtrent le sujet. Comme mentionné précédemment, de nombreuses techniques d'indexage existent, alors pourquoi vouloir en inventer une nouvelle ?

L'algorithme décrit dans (Gräf 1991) permet d'obtenir une implantation efficace en utilisant des automates déterministes, qui ne remettent jamais en cause les transitions effectuées. La construction de ces automates est complexe et ils sont généralement composés d'un très grand nombre d'états qui peut limiter leur utilisation. Un autre algorithme permettant de réduire ce nombre d'états en mettant en facteur certaines parties redondantes de l'automate est présenté dans (Nedjah 1997, Nedjah et al. 1997). Mais ici encore, la procédure chargée de reconnaître les états à mettre en facteur est particulièrement coûteuse et complexe à mettre en œuvre. De plus, ces travaux n'étaient pas encore publiés lorsque nous avons débuté notre recherche, et c'est pouquoi nous avons développé, en parallèle, notre propre algorithme permettant de réduire le nombre d'états des automates.

Les autres approches telles que (Sekar et al. 1992, Christian 1993, Graf 1996) sont intéressantes mais ne sont pas forcément adaptées à l'utilisation que nous voulons en faire. Ces algorithmes permettent de construire efficacement des automates de filtrages, mais ces automates ne sont pas déterministes, ce qui, dans l'approche compilée, est un handicap. Ils sont généralement mieux adaptés à la réalisation d'outils de déduction automatique où l'ensemble des règles (appelé aussi « base de connaissances ») est amené à se modifier dynamiquement au cours des calculs.

Dans ce chapitre, nous présentons la notion d'automate de filtrage et nous nous intéressons particulièrement aux automates dits *déterministes*. Pour un ensemble de motifs donnés, la construction de tels automates conduit généralement à calculer une *clôture* de l'ensemble de motifs, ce qui augmente considérablement le nombre d'états de l'automate associé. Dans un premier temps, nous proposons un calcul incrémental de cette clôture, puis nous dérivons un algorithme plus simple permettant de construire une clôture « réduite ». L'introduction de telles clôtures « réduites » permet en particulier de réduire la taille des automates engendrés en factorisant un grand nombre d'états. En contre-partie, les automates associés ne permettent (temporairement) plus de reconnaître certains motifs de l'ensemble initial. Pour pallier cet inconvénient, nous introduisons la notion de jumpNode, ce qui nous permet de construire des automates efficaces, déterministes et de taille relativement petite.

5.1 Termes vus comme des chaînes de symboles

L'algorithme de filtrage présenté dans ce chapitre est plus facile à exprimer et à comprendre lorsqu'on utilise une représentation particulière des termes. Les termes sont souvent vus comme des arbres où il y a une correspondance directe entre le symbole de tête du terme et la racine de l'arbre d'une part ; les sous-termes de ce symbole et les sous-arbres de la racine d'autre part. Dans ce chapitre, nous proposons de voir le terme sous sa forme «aplatie». Sa structure ressemble alors plus à une chaîne de symboles qu'à un arbre.

Étant donné un alphabet non vide Σ, à chaque symbole s de Σ est associé un entier positif ou nul, appelé arité et noté $\#s$. On suppose généralement qu'il existe un symbole $\omega \in \Sigma$ d'arité 0 qui joue le rôle de symbole de variable anonyme. L'ensemble des chaînes construites à partir de Σ est noté Σ^*, il contient en particulier la chaîne vide (ne contenant aucun symbole) notée ϵ. La longueur d'une chaîne $\alpha \in \Sigma^*$ donnée se note $|\alpha|$ et correspond au nombre de symboles la composant.

L'ensemble $\mathcal{T}_\Sigma$ des termes construits sur Σ est le plus petit sous-ensemble de Σ^* contenant les chaînes « bien formées » $st_1 \dots t_n$ telles que $s \in \Sigma$, $n \in \mathbb{N}$, $\#(s) = n$ et $t_1, \dots, t_n \in \mathcal{T}_\Sigma$.

5.2 Automate de filtrage

Étant donné un terme clos t et un ensemble de termes $L = \{t_1, \dots, t_n\}$ (aussi appelés *motifs*), nous voulons définir une procédure de décision dont le résultat est le plus grand sous-ensemble $R \subseteq L$ tel que t soit une instance de tous les éléments de R. Si R est l'ensemble vide, on dit que la procédure échoue et qu'aucun motif n'est reconnu. Notre approche consiste à définir une telle procédure de décision par un système de transition à états finis dont les règles de transition dépendent de l'ensemble initial L. Ce système de transition, appelé *automate de filtrage*, dispose d'une tête de lecture qui peut se déplacer de gauche à droite sur une suite de symboles donnée en entrée. Cette suite est la représentation sous forme de chaîne du terme clos que l'on veut reconnaître. L'automate de filtrage est assez particulier dans la mesure où sa tête de lecture peut se déplacer de deux manières différentes :

- elle peut lire un symbole et se déplacer d'un cran vers la droite ;
- elle peut aussi lire un terme complet (bien formé) et se déplacer vers la droite d'autant de symboles que nécessaire.

Un automate de filtrage A est défini par un tuple :

$$A = (\Sigma, E, e_0, F, \Delta)$$

où

1. Σ est un alphabet ;
2. E est un ensemble fini d'états ;
3. $e_0 \in E$ est l'état initial ;
4. $F \subseteq E$ est l'ensemble des états finaux ;
5. $\Delta = \{\delta_1, \dots, \delta_n\}$ est l'ensemble des *règles de transition d'états* de A, où les $\delta_i : E \times \{\Sigma \cup \mathcal{T}_\Sigma\} \mapsto E$ sont de la forme $(e, s) \longrightarrow_\delta e'$, avec $e, e' \in E$ et $s \in \Sigma$.

Les règles de transition d'états décrivent la *structure* de l'automate en énumérant tous les changements d'états possibles. Elles sont utilisées pour autoriser ou non le passage d'un état vers un autre. Les règles décrivant le *comportement* de l'automate (le déplacement de la tête de lecture) sont appelées *règles de transition de configurations* $\longrightarrow_R$. Elles coordonnent les changements d'états de l'automate et la position de la tête de lecture sur la bande d'entrée. Les règles de transition de configurations sont définies par :

$$\begin{aligned} (e, s\alpha) &\longrightarrow_R (e_1, \alpha) \quad \text{ssi} \quad s \in \Sigma \setminus \{\omega\} \text{ et } (e, s) \longrightarrow_\delta e_1 \in \Delta \\ (e, \alpha\beta) &\longrightarrow_R (e_2, \beta) \quad \text{ssi} \quad \alpha \in \mathcal{T}_\Sigma \text{ et } (e, \omega) \longrightarrow_\delta e_2 \in \Delta \end{aligned}$$

On dit que l'automate de filtrage A reconnaît un terme $t \in \mathcal{T}_\Sigma$ s'il existe une suite de transitions qui mène l'automate dans un état final et que le terme d'entrée a été complètement lu :

$$\exists e \in F \text{ tel que } (e_0, t) \xrightarrow{*}_R (e, \epsilon)$$

Étant donné un ensemble de termes $L = \{t_1, \dots, t_n\} \subseteq \mathcal{T}_\Sigma$, on dit que l'automate de filtrage reconnaît le langage L si toute instance d'un terme de L est reconnue par A. Un exemple d'automate de filtrage reconnaissant le langage $L = \{fga, f\omega\}$ est donné sur la figure 5.1.

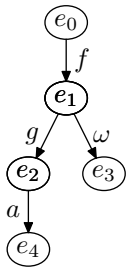


FIG. 5.1 – *Cet automate reconnaît le langage* $L = \{fga, f\omega\}$.

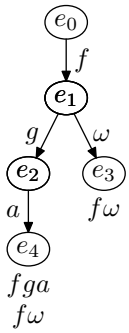


FIG. 5.2 – *Les états terminaux de cet automate sont décorés afin d'implanter une procédure de décision pour le langage* $L = \{fga, f\omega\}$.

Partant d'un ensemble de motifs L, il est facile de construire un tel automate : un état initial est créé et pour chaque ensemble de termes commençant par un même symbole de tête, un nouvel état et une règle de transition d'états sont créés. L'algorithme s'applique ensuite récursivement sur les sous-termes. L'automate ainsi construit n'est généralement pas *déterministe* dans la mesure où deux configurations finales différentes peuvent être atteintes en partant d'une même configuration initiale.

Rappelons notre objectif initial : il s'agit de définir une procédure de décision pour un langage L. Étant donné un terme clos t, cette procédure doit retourner le plus grand sous-ensemble $R \subseteq L$ tel que t soit une instance de tous les éléments de R.

L'automate précédemment construit peut être utilisé pour implanter une telle procédure de décision : il suffit de « décorer » les états finaux de l'automate par un ensemble de motifs reconnus (voir figure 5.2). L'état e_4 se voit ainsi décoré par les motifs $\{fga, f\omega\}$ et l'état e_3 par l'unique motif $\{f\omega\}$.

Pour reconnaître fga il suffit de suivre les arêtes reliant deux états en respectant les règles de transition de configurations. Partant de l'état e_0, le symbole f est lu et on se retrouve dans l'état e_1 avec le *suffixe* ga restant à lire. Le problème se corse parce qu'il y a un choix à faire : quelle arête suivre ? Les règles de changement de configurations disent que nous pouvons suivre les deux : lire g puis a et se retrouver dans l'état e_4 ou bien lire directement le terme bien formé ga et s'arrêter dans l'état e_3. Il faudrait avoir deux têtes de lecture pour pouvoir explorer les deux possibilités en parallèle, mais ne disposant pas de ce luxe, l'automate doit suivre une branche et consommer les deux symboles g, a avant de les remettre sur la bande de lecture pour pouvoir

explorer la branche restante.

L'automate n'étant pas déterministe, il faut explorer tous les états finaux atteignables pour être sûr de retourner le plus grand sous-ensemble R (R se compose des différentes « décorations » rencontrées au cours de l'exploration). C'est ce manque de déterminisme mettant en œuvre une stratégie avec des *retours arrières* qui nous déplaît particulièrement. En effet, le terme d'entrée doit souvent être inspecté autant de fois que le nombre de termes composant L, ce qui est une source d'inefficacité.

Pour éviter cette exploration laborieuse, une autre solution consiste à construire un automate ne pouvant atteindre qu'une seule et unique configuration à partir d'une configuration de départ. Ce type d'automate, n'effectuant aucun retour arrière, est dit *déterministe*.

L'idée derrière ces automates est la suivante : pour chaque état $e \in E$ et pour chaque symbole $s \in \Sigma$ il ne doit pas y avoir plus d'une règle de transition d'états $(e,s) \longrightarrow_\delta e' \in \Delta$. Mais ces contraintes ne suffisent pas à construire l'automate recherché. En effet, l'automate de la figure 5.1 respectait déjà ces contraintes sans pour autant s'arrêter dans un unique état. Un tel automate est dit *faiblement déterministe*.

Le non-déterminisme est ainsi réduit aux situations où l'automate a le choix entre « lire un symbole et suivre une arête étiquetée par le même symbole » ou « lire un terme bien formé et suivre une arête étiquetée par un ω ». Afin d'éliminer complètement ces situations de choix, nous allons modifier légèrement les règles du jeu en interdisant d'utiliser une arête étiquetée par un ω s'il y a une autre alternative possible. Les *règles de transition de configurations canoniques* se définissent de la manière suivante :

$$(e,s\alpha\beta) \xrightarrow[can]{} R \begin{cases} (e_1,\alpha\beta) & \text{si} \quad s \in \Sigma \setminus \{\omega\} \text{ et } (e,s) \longrightarrow_\delta e_1 \in \Delta \\ (e_2,\beta) & \text{sinon,} \quad \text{et si } s\alpha \in \mathcal{T}_\Sigma \text{ et } (e,\omega) \longrightarrow_\delta e_2 \in \Delta \end{cases}$$

Un automate faiblement déterministe utilisant ces règles de transition de configurations canoniques est dit *déterministe* : les transitions étiquetées par ω ne peuvent être appliquées que s'il n'existe pas d'autre transition étiquetée par $s \neq \omega$. Ainsi, pour une entrée α et un état e_i, il y a au plus une seule chaîne de transitions $(e_i,\alpha) \xrightarrow[can]{*} R\ (e_j,\beta)$ avec $e_i,e_j \in E$. Il peut cependant arriver qu'un terme t soit reconnu par un automate de filtrage faiblement déterministe, mais pas par sa version déterministe. Reprenons l'automate faiblement déterministe de la figure 5.1, on avait bien $(e_0,fgb) \xrightarrow{*} R\ (e_3,\epsilon)$ et pourtant, en utilisant les règles de transition canoniques, on ne peut construire que la chaîne $(e_0,fgb) \xrightarrow[can]{*} R\ (e_2,b)$ avec $e_2 \notin F$. Le terme fgb n'est donc pas reconnu.

Étant donné un automate faiblement déterministe A, il est dit *canonique* si tout terme reconnu par A est aussi reconnu par sa version déterministe. Le contre-exemple précédent montre que l'automate considéré n'est pas *canonique*, il peut cependant le devenir si on étend le langage L en lui ajoutant le terme $fg\omega$. Considérons $\overline{L} = L \cup \{fg\omega\} = \{fga,fg\omega,g\omega\}$, appelé *clôture* de l'ensemble L. Il est maintenant facile de vérifier que l'automate associé est bien canonique (voir figure 5.3).

La partie délicate dans l'algorithme de construction d'une procédure de décision pour un langage donné L n'est pas tellement la construction de l'automate déterministe mais principalement le calcul de la clôture $\overline{L}$ à considérer pour obtenir un automate canonique. Dans les paragraphes suivants, nous allons étudier différents algorithmes permettant de construire la clôture $\overline{L}$ à partir d'un ensemble de termes L.

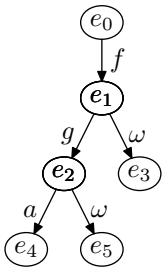


FIG. 5.3 – *Cet automate est déterministe et reconnaît le langage* $L = \{fga, fg\omega, f\omega\}$ *en n'inspectant qu'une seule fois les termes d'entrée.*

5.3 Clôtures d'un ensemble de motifs

Étant donné en ensemble de termes $L = \{t_1, \ldots, t_n\}$, pour calculer sa clôture $\overline{L}$ nous permettant de dériver facilement un automate canonique reconnaissant le langage L, nous avons besoin de réaliser des opérations sur des ensembles de *suffixes* de termes. Soit $\alpha\beta \in \mathcal{T}_\Sigma$, α et β sont des suites de symboles ($\alpha, \beta \in \Sigma^*$), mais pas forcément des termes bien formés. Dans ce cas, α et β sont appelés respectivement *préfixe* et *suffixe* du terme $\alpha\beta$. Soit $t \in \mathcal{T}_\Sigma$, les ensembles de préfixes et suffixes de t sont définis de la manière suivante :

$$\mathsf{Pref}(t) = \{\alpha \mid \alpha \in \Sigma^* \text{ et } \exists \beta \in \Sigma^* \text{ tel que } \alpha\beta = t\}$$

$$\mathsf{Suff}(t) = \{\beta \mid \beta \in \Sigma^* \text{ et } \exists \alpha \in \Sigma^* \text{ tel que } \alpha\beta = t\}$$

Considérons maintenant un ensemble de suffixes L (initialement, les suffixes sont des termes bien formés) et un symbole s. L'ensemble de suffixes obtenu en enlevant le symbole de tête s des éléments de L qui commencent par s est noté L/s. Dans son article, Albert Gräf (1991) donne un algorithme récursif pour calculer la clôture d'un ensemble de suffixes :

$$\overline{L} = \begin{cases} L & \text{si } L = \{\epsilon\} \text{ ou } L = \emptyset \\ \bigcup_{s \in \Sigma \cup \{\omega\}} s\overline{L_s} & \text{sinon} \end{cases}$$

où L_s est défini de la manière suivante ($\omega^{\#s}$ correspond à la répétition de $\#s$ symboles ω) :

$$L_s = \begin{cases} L/s & \text{si } s = \omega \\ L/s \cup \omega^{\#s} L/\omega & \text{si } s \neq \omega \text{ et } L/s \neq \emptyset \\ \emptyset & \text{sinon} \end{cases}$$

La clôture d'un ensemble a les propriétés suivantes :

$$\begin{array}{ll} \textit{extension} & L \subseteq \overline{L} \\ \textit{monotonie} & L \subseteq M \Longrightarrow \overline{L} \subseteq \overline{M} \\ \textit{idempotence} & \overline{\overline{L}} = \overline{L} \end{array}$$

On voit, de manière intuitive, que cet algorithme ajoute des suffixes afin que l'automate associé ne se « bloque » plus dans une branche. Étant donnés deux suffixes $s\beta$ et $\omega\beta'$, on sait que l'automate va choisir la branche étiquetée par s même si la suite du terme d'entrée se termine

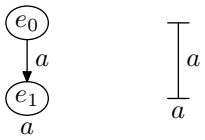


FIG. 5.4 – *Cet exemple montre deux représentations possibles de l'ensemble $L = \{a\}$: un automate de filtrage (à gauche) et un arbre de filtrage (à droite).*

par β'. Pour éliminer ces situations d'échec et retarder le choix entre ces deux alternatives, le suffixe $s\omega^{\#^s}\beta'$ est ajouté à l'ensemble.

Appliquons l'algorithme précédent sur l'ensemble $L = \{fga,f\omega\}$ pour vérifier que sa clôture $\overline{L}$ est bien égale à $\{fga,fg\omega,f\omega\}$:

$$\begin{array}{lcl}
\overline{L} & = & f\overline{L_f} \cup g\overline{L_g} \cup a\overline{L_a} \cup \omega\overline{L_\omega} \\
 & = & f\overline{L_f} \quad (\text{car } L_g = L_a = L_\omega = \emptyset) \\
\overline{L_f} & = & \overline{\{ga,\omega\}} \\
 & = & g\overline{(L_f)_g} \cup \omega\overline{(L_f)_\omega} \\
 & = & g\overline{\{\{a\} \cup \omega^{\#_g} L_f/\omega\}} \cup \omega\overline{\{\epsilon\}} \\
 & = & g\{a,\omega\} \cup \{\omega\} \\
 & = & \{ga,g\omega,\omega\} \\
\text{d'où} & & \\
\overline{L} & = & \{fga,fg\omega,f\omega\}
\end{array}$$

Le calcul d'une clôture est un peu ennuyeux mais la grande utilité de l'ensemble ainsi produit justifie pleinement notre intérêt. Albert Gräf montre en effet que pour tout ensemble fini de termes $L \subseteq \mathcal{T}_\Sigma$, l'automate déterministe associé à $\overline{L}$ est bien canonique (il reconnaît le langage L).

À cet endroit, le lecteur doit s'assurer qu'il perçoit bien les liens existant entre la clôture d'un ensemble de termes et l'automate canonique capable de reconnaître toute instance de cet ensemble. Ces deux objets doivent être vus comme deux représentations différentes de la même idée : modifier l'ensemble de termes c'est modifier l'automate associé, mais effectuer des changements sur l'automate c'est aussi changer l'ensemble de termes reconnu. Dans la pratique, il est souvent plus facile de manipuler directement un automate, dont la structure se prête mieux au traitement informatique que des ensembles de suffixes. Il existe cependant une alternative consistant à représenter des ensembles de suffixes par des arbres. On parle alors *d'arbre de filtrage*. Un arbre de filtrage est défini moins formellement qu'un automate mais lui ressemble beaucoup : les états de l'automate ne sont plus nommés et deviennent des *nœuds*, les règles de transition d'états deviennent des arêtes reliant deux nœuds (voir figure 5.4).

Afin de mettre en évidence les liens existant entre ces trois notions nous allons présenter un lemme (établi dans (Gräf 1991)) et en dériver un algorithme incrémental de construction d'arbre de filtrage. Cette approche évite d'avoir à calculer une clôture pour en dériver un automate de filtrage, mais au contraire, un arbre de filtrage est directement construit et la clôture associée est une conséquence de cette construction.

Définissons l'opérateur ∇ comme étant la clôture de l'union de deux ensembles clos : soient M et N deux ensembles clos (i.e. $M = \overline{M}$ et $N = \overline{N}$),

$$\nabla(M,N) = \overline{(M \cup N)}$$

En particulier on a :$\nabla(\emptyset,N) = N$, $\nabla(M,\emptyset) = M$, et si $M = N = \{\epsilon\}$, alors $\nabla(M,N) = \overline{M \cup N} = M \cup N = \{\epsilon\}$. Supposons maintenant que $M \cup N \not\subseteq \{\epsilon\}$. D'après (Gräf 1991),

$$\overline{M \cup N} = \bigcup_{s \in \Sigma} s(\overline{M_s \cup N_s})$$

où $M_\omega = M/\omega$, $N_\omega = N/\omega$ et pour tout $s \neq \omega$:

$$M_s = \begin{cases} M/s & \text{si } M/s \neq \emptyset \\ \omega^{\#s} M/\omega & \text{si } M/s = \emptyset \text{ et } M/\omega \neq \emptyset \text{ et } N/s \neq \emptyset \\ \emptyset & \text{sinon} \end{cases}$$

$$N_s = \begin{cases} N/s & \text{si } N/s \neq \emptyset \\ \omega^{\#s} N/\omega & \text{si } N/s = \emptyset \text{ et } N/\omega \neq \emptyset \text{ et } M/s \neq \emptyset \\ \emptyset & \text{sinon} \end{cases}$$

On peut remarquer que le calcul de M_s dépend de l'ensemble N et réciproquement.

Nous voulons définir un algorithme pour construire de manière incrémentale la clôture ou l'arbre de filtrage recherché. Partant d'un ensemble clos L, nous ajoutons les suffixes un à un, et en fonction de la structure du motif, différents types d'insertion sont effectués. En supposant que le suffixe à insérer est de la forme $\{sp\}$, avec $p \in \Sigma^*$, on a alors $\overline{\{sp\}} = \{sp\}$ et nous pouvons utiliser le lemme précédent pour calculer les valeurs des ensembles suivants :

$$\begin{array}{lcl} \{sp\}_s & = & \{p\} \\ \{\omega p\}_s & = & \{\omega^{\#s} p\} \text{ si } L/s \neq \emptyset \\ \{s' p\}_s & = & \emptyset \text{ pour } s' \neq s \end{array}$$

Pour calculer $\nabla(L,\{sp\}) = \overline{L \cup \{sp\}}$, nous distinguons deux cas :

1. le suffixe ajouté commence par un symbole qui n'est pas une variable : $s \neq \omega$

$$\begin{array}{rcl} \nabla(L,\{sp\}) & = & \displaystyle\bigcup_{s' \in \Sigma} s'(\overline{L_{s'} \cup \{sp\}_{s'}}) \\ & = & \displaystyle\bigcup_{s' \neq s} s' \nabla(L_{s'},\{sp\}_{s'}) \cup s \nabla(L_s,\{sp\}_s) \\ & = & \displaystyle\bigcup_{s' \neq s} s' \nabla(L_{s'},\emptyset) \cup s \nabla(L_s,\{p\}) \\ & = & \displaystyle\bigcup_{s' \neq s} s' L_{s'} \cup s \nabla(L_s,\{p\}) \end{array}$$

Il faut encore distinguer deux possibilités :

- $L_\omega = \emptyset$ ou $L/s \neq \emptyset$: dans ce cas on a $L_s = L/s$. Comme $\bigcup_{s' \neq s} s' L_{s'} \subseteq L \subseteq \nabla(L,\{sp\})$, on en déduit :

$$\boxed{\nabla(L,\{sp\}) = L \cup s\nabla(L/s,\{p\})}$$

  Partant de l'ensemble L, cela revient à ajouter récursivement le suffixe $\{p\}$ au sous ensemble L/s (qui peut être éventuellement vide).
- $L_\omega \neq \emptyset$ et $L/s = \emptyset$: dans ce cas on a $L_s = \omega^{\#s} L_\omega$ et

$$\boxed{\nabla(L,\{sp\}) = L \cup s\nabla(\omega^{\#s} L_\omega,\{p\})}$$

  Ce cas est un peu plus compliqué : le sous-ensemble $\omega^{\#s} L_\omega$ est créé, puis le suffixe $\{p\}$ lui est ajouté. L'ensemble étant évidemment ajouté à L.

2. le suffixe ajouté commence par une variable : $s = \omega$

$$\begin{aligned}
\nabla(L,\{\omega p\}) &= \bigcup_{s'\in\Sigma} s'(\overline{L_{s'} \cup \{\omega p\}_{s'}}) \\
&= \bigcup_{s'\neq\omega} s'\nabla(L_{s'},\{\omega p\}_{s'}) \cup \omega\nabla(L_\omega,\{\omega p\}_\omega) \\
&= \bigcup_{\substack{s'\neq\omega \\ L/s'\neq\emptyset}} s'\nabla(L_{s'},\{\omega^{\#s'} p\}) \cup \omega\nabla(L_\omega,\{p\}) \bigcup_{\substack{s'\neq\omega \\ L/s'=\emptyset}} s'\nabla(\emptyset,\emptyset) \\
&= \bigcup_{\substack{s'\neq\omega \\ L/s'\neq\emptyset}} s'\nabla(L_{s'},\{\omega^{\#s'} p\}) \cup \omega\nabla(L_\omega,\{p\})
\end{aligned}$$

L'opérateur de clôture étant monotone, on peut ajouter l'ensemble L à l'ensemble résultat, ce qui nous donne :

$$\boxed{\nabla(L,\{\omega p\}) = L \cup \bigcup_{\substack{s'\neq\omega \\ L/s'\neq\emptyset}} s'\nabla(L_{s'},\{\omega^{\#s'} p\}) \cup \omega\nabla(L_\omega,\{p\})}$$

Cela correspond à une insertion en deux étapes : le suffixe $\{p\}$ est dans un premier temps ajouté à la sous-branche étiquetée par un ω ($L_\omega = L/\omega$). Dans un deuxième temps, le suffixe $\{\omega^{\#s'} p\}$ est inséré dans tous les sous-arbres L/s', $s' \neq \omega$ ($L_{s'} = L/s'$ car $L/s' \neq \emptyset$).

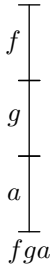
Les expressions trouvées, correspondant aux différentes valeurs possible de $\nabla(L,\{sp\})$, nous permettent de dériver un algorithme de calcul incrémental de clôtures (Algorithme 5.1). En même temps que la clôture se construit, les motifs sont organisés dans une structure arborescente.

Algorithme 5.1 Calcul incrémental d'un arbre de filtrage correspond à la clôture d'un ensemble de suffixes

```
1: ajout du suffixe {sp} =
2: si s ≠ ω alors
3:    si L_ω = ∅ ou L/s ≠ ∅ alors
4:       le suffixe {p} est ajouté à L/s
5:    sinon
6:       l'arête étiquetée par s est créée, puis les suffixes {p} et ω^{#s} L_ω sont ajoutés à L/s
7:    finsi
8: sinon
9:    si ω est un nouveau choix possible, une arête étiquetée par ω est créée
10:   le suffixe {p} est ajouté à L/ω
11:   pour tout symbole s' ≠ ω tel que L/s' ≠ ∅ faire
12:      les suffixes ω^{#s'} p sont ajoutés à L/s'
13:   fin pour
14: finsi
```

Exemple 1 *Soit $L = \{fga, f\omega\}$. L'algorithme 5.1 permet de construire la clôture $\overline{L}$ de manière incrémentale : les motifs fga et $f\omega$ sont insérés l'un après l'autre. L'ordre d'insertion n'a aucune influence sur le résultat, c'est pourquoi dans la pratique, ils sont insérés au fur et à mesure de leur définition. Commençons par fga : l'arbre de filtrage étant initialement vide, l'arête étiquetée*

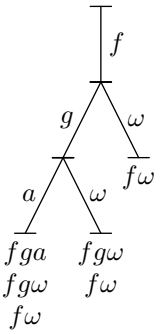
par f est créée puis le suffixe ga est récursivement ajouté au sous-arbre vide. On obtient alors l'arbre suivant :



Le deuxième motif $f\omega$ est ensuite inséré : l'arête étiquetée par f est suivie puis le suffixe ω est ajouté au sous-arbre courant. Il s'agit d'un suffixe commençant par un ω. Son insertion fait appel à la partie la plus complexe de l'algorithme :

- *une arête étiquetée par ω est créée, puis la feuille de l'arbre est décorée par le motif $f\omega$ pour indiquer qu'il a été reconnu (voir partie droite de la figure suivante) ;*
- *les autres arêtes sont suivies (il n'y en a qu'une, celle étiquetée par g) et le suffixe $\omega^{\#g} = \omega$ est ajouté. Une nouvelle fois, il s'agit d'ajouter un suffixe commençant par une variable : une arête ω est créée, sa feuille est décorée par deux motifs ($fg\omega$ et $f\omega$), et récursivement, ces décorations se propagent vers la feuille la plus à gauche.*

L'arbre suivant est finalement obtenu :



5.4 Clôture « réduite » d'un ensemble de motifs

La taille des arbres construits par l'algorithme 5.1 devient rapidement importante, voire ingérable lorsque l'ensemble initial de motifs contient un grand nombre de « recouvrements de préfixes ». Un ensemble de motifs L est dit avec « recouvrements de préfixes » s'il existe un préfixe clos $\alpha \in \Sigma^*$ tel qu'il existe deux motifs distincts de L dont les préfixes filtrent vers α. Considérons, par exemple, l'ensemble $L = \{fab, f\omega c\}$, où f est un symbole d'arité 2. L est avec recouvrements de préfixes parce qu'il existe un préfixe clos $\alpha = fa$ qui est une instance des deux préfixes $fa \in \mathsf{Pref}(fab)$ et $f\omega \in \mathsf{Pref}(f\omega c)$.

En effet, lorsqu'un suffixe commençant par une variable ω est inséré dans un arbre de filtrage, cela provoque l'ajout d'un suffixe commençant par un certain nombre de variables : $\omega^{\#s'}p$. Un effet « boule de neige » se produit et de nombreux suffixes sont ajoutés en cascade.

Dans la suite de ce chapitre nous proposons un nouvel algorithme de construction de « clôtures réduites » qui diminue considérablement la taille des arbres construits. L'algorithme se décompose en deux étapes :

1. la première phase construit une « clôture réduite » (Algorithme 5.2) : les suffixes $\omega^{\#s'}p$ qui provoquaient des ajouts en cascades ne sont plus ajoutés. La clôture obtenue n'est évi-

demment plus équivalente à celle calculée par les algorithmes précédents, et en particulier, l'automate déterministe associé n'est plus canonique. L'automate ainsi construit est dit *faiblement canonique*.

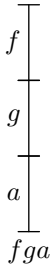
2. la deuxième phase de l'algorithme ajoute de nouvelles règles de transition d'états à l'automate pour simuler l'ajout des suffixes « oubliés ». Ces règles particulières de transition d'états, appelées jumpNode, permettent de réduire le nombre d'états de l'automate construit tout en conservant ses propriétés déterministes. D'un point de vue observationnel, l'automate *faiblement canonique*, associé à la « clôture réduite », enrichi par de nouvelles règles de transition d'états (jumpNode), a exactement le même comportement que l'automate construit par l'algorithme 5.1. Il devient donc canonique.

Algorithme 5.2 Calcul incrémental d'un arbre de filtrage correspondant à une « clôture réduite »

ajout de $\{sp\}$ = `/* le suffixe` $\{sp\}$ `peut aussi s'écrire` $\{st_1 \dots t_{\#_s} p'\}$ `*/`
si $s \neq \omega$ **alors**
  si $L_\omega = \emptyset$ ou $L/s \neq \emptyset$ **alors**
    le suffixe $\{p\}$ est ajouté à L/s
  sinon
    l'arête étiquetée par s est créée, puis les suffixes $\{p\}$ et $t_1 \dots t_{\#_s} L_\omega$ sont ajoutés à L/s `/* on peut remarquer que les` $t_1 \dots t_{\#_s} L_\omega$ `sont des instances de` $\omega^{\#_s} L_\omega$ `*/`
  finsi
sinon
  si ω est un nouveau choix possible, une arête étiquetée par ω est créée
  le suffixe $\{p\}$ est ajouté à L/ω
  pour tout symbole $s' \neq \omega$ tel que $L/s' \neq \emptyset$ **faire**
    pour tout $\beta = t_1 \dots t_{\#_{s'}}$ préfixe de $L_{s'}$ ($s'\beta$ est un terme bien formé) **faire**
      le suffixe $\beta p'$ est ajouté à $L_{s'}$ `/* on peut remarquer que les` βp`, instances de` $\omega^{\#_{s'}} p$`, ne sont plus ajoutées par cet algorithme */`
    fin pour
  fin pour
finsi

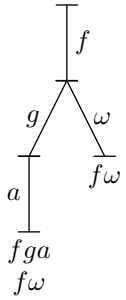
Exemple 2 *Soit* $L = \{fga, f\omega\}$. *L'algorithme 5.2 permet de construire une « clôture réduite »* $\widetilde{L}$ *contenant généralement moins d'éléments que* $\overline{L}$. *En particulier, on a :* $L \subseteq \widetilde{L} \subseteq \overline{L}$.

Comme dans l'exemple 1, partant d'un arbre de filtrage vide, l'insertion du premier terme fga *construit l'arbre suivant :*



Le deuxième motif $f\omega$ *est ensuite inséré : l'arête étiquetée par* f *est suivie puis la branche étiquetée par* ω *et décorée par* $f\omega$ *est créée. L'arête étiquetée par* g *est alors suivie, mais à la*

différence de l'exemple 1, l'insertion du ω est simplifiée : seule son instance a est réinsérée et la feuille gauche de l'arbre est décorée par $f\omega$. On obtient alors :



Cet arbre est plus « petit » que celui de l'exemple 1, l'automate associé n'est plus canonique mais faiblement canonique. On peut remarquer que sur cet exemple, l'automate produit est le même que celui de la figure 5.2 (automate nécessitant une stratégie de retour arrière pour implanter la procédure de décision recherchée). Un phénomène identique à celui décrit au paragraphe 5.2 se produit : le terme clos fgb n'est plus reconnu par l'automate, car après avoir lu le f et le g, il est « trop tard » pour changer de branche.

L'exemple 2 met en évidence l'insuffisance de la « clôture réduite » construite par l'algorithme 5.2. Nous avons simplifié le calcul de clôture d'Albert Gräf, dérivé un algorithme incrémental pour construire directement un arbre de filtrage, puis modifié légèrement l'algorithme pour construire une « clôture réduite », mais en perdant l'aspect canonique de l'automate. Il est naturel de se demander si l'automate ne deviendrait pas canonique, si au moment de lire le b (de fgb) on pouvait remettre en cause le dernier choix effectué et « sauter » vers la branche étiquetée par ω.

5.5 Automate de filtrage à mémoire

Dans cette partie nous considérons des *automates de filtrage à mémoire* ressemblant beaucoup à ceux définis au paragraphe 5.2. Une des différences réside dans le mode de déplacement de la tête de lecture. Celle-ci peut maintenant se déplacer de trois manières différentes :

- elle peut lire un symbole et se déplacer d'un cran vers la droite ;
- elle peut lire un terme complet et se déplacer vers la droite d'autant de symboles que nécessaire ;
- elle peut aussi, étant donné un préfixe α déjà lu, lire le suffixe β correspondant (tel que le terme $\alpha\beta$ soit bien formé) et se déplacer vers la droite d'autant de symboles que nécessaire.

Un automate de filtrage à mémoire A est défini par un tuple :

$$A = (\Sigma, E, e_0, F, \Delta, \mathcal{M})$$

où

1. Σ est un alphabet ;
2. E est un ensemble fini d'états ;
3. $e_0 \in E$ est l'état initial ;
4. $F \subseteq E$ est l'ensemble des états finaux ;
5. $\Delta = \{\delta_1, \ldots, \delta_n\}$ est l'ensemble des *règles de transition d'états* de A, où les $\delta_i : E \times \{\Sigma \cup \mathcal{T}_\Sigma\} \mapsto E$ sont de la forme $(e,s) \longrightarrow_\delta e'$, avec $e, e' \in E$ et $s \in \Sigma$;

6. $\mathcal{M}$ est une mémoire permettant de mémoriser un ensemble de couples $(e,\alpha) \in E \times \Sigma^*$ (e correspond à un état où un choix a été fait et α correspond au préfixe lu sur la bande depuis cet état).

Au paragraphe 5.2, nous avons présenté des automates de filtrage (non déterministes) en donnant des règles de transition de configurations volontairement « ambiguës ». Il fallait utiliser une *stratégie* de recherche avec retour arrière pour que l'ensemble des états finaux puisse être exploré. Mais cette stratégie, permettant de restaurer un état particulier de la bande de lecture, ne faisait pas partie du système : sa description n'était pas contenue dans celle de l'automate. Et pourtant, il fallait bien une « mémoire » particulière pour se souvenir des branches non explorées. L'introduction d'automates de filtrage à mémoire a pour objectif de clarifier et d'expliciter, dans le système lui-même, une forme particulière de stratégie avec retour arrière.

La stratégie est « particulière » dans la mesure où nous ne voulons pas « rembobiner » la bande de lecture, ce qui reviendrait à déplacer la tête de lecture vers la gauche et donc à lire plusieurs fois le terme d'entrée. C'est principalement ce que nous voulons éviter pour des raisons évidentes d'efficacité. Lorsque l'automate a le choix entre suivre une arête étiquetée par un symbole s ou suivre une arête étiquetée par un ω, il choisit toujours de suivre celle qui est étiquetée par le symbole $s \neq \omega$, mais il mémorise, dans sa mémoire $\mathcal{M}$ prévue à cet effet, l'état où s'est fait le choix et, par la suite, tous les symboles lus depuis cet état. Lorsque l'automate se « bloque » dans un état qui n'est pas terminal, c'est peut-être qu'un mauvais choix a été fait dans le passé. L'automate regarde alors la position de sa tête de lecture, puis inspecte sa mémoire à la recherche du dernier choix effectué tel que si l'arête ω avait été utilisée, cela aurait permis à la tête de lecture d'atteindre un point plus à droite sur la bande de lecture. Si un tel état e est trouvé dans la mémoire $\mathcal{M}$, un préfixe α lui est associé, le troisième mode de déplacement de la tête de lecture est alors utilisé : celle-ci avance vers la droite en lisant autant de symboles $\beta \in \Sigma^*$ nécessaires pour que $\alpha\beta$ soit un terme bien formé. L'automate utilise alors la règle de transition d'états δ_i : $(e,\omega) \longrightarrow_\delta e'$ pour passer dans l'état e' (c'est la règle qui n'avait pas été utilisée lorsque le « mauvais » choix a été fait).

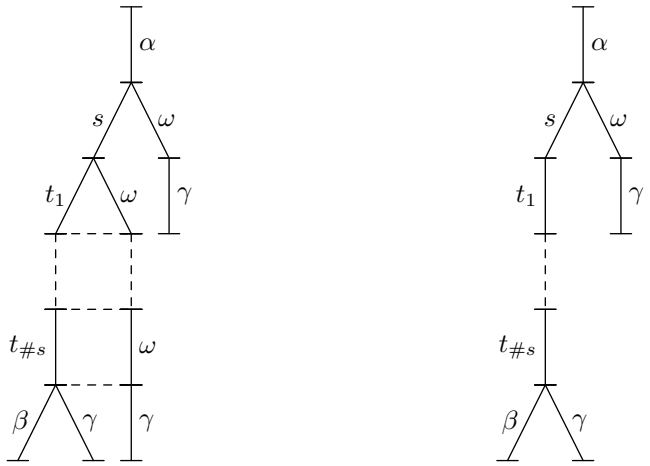
La stratégie mise en œuvre est dite *avec retours arrière* parce que l'automate retourne dans un état passé pour effectuer un autre choix. La particularité de cette stratégie est qu'elle ne fait jamais reculer la tête de lecture, au contraire, elle a tendance à la faire se déplacer plus rapidement vers la droite. Ce déplacement unidirectionnel assure que le terme d'entrée n'est inspecté qu'une seule fois pour déterminer s'il appartient au langage reconnu par l'automate.

Nous avons vu qu'étant donné un ensemble de motifs L, à partir de sa clôture $\overline{L}$ on peut facilement construire un automate canonique reconnaissant le langage L. L'inconvénient de cette approche est que l'automate ainsi construit comporte un nombre généralement important d'états. Une autre approche consiste à calculer la clôture réduite $\widetilde{L} \subseteq \overline{L}$ et à dériver l'automate faiblement canonique associé. Nous avons vu que la version « sans mémoire » de cet automate faiblement canonique ne reconnaît pas le langage L, mais nous allons voir que la version « à mémoire » est bien équivalente à l'automate canonique construit à partir de la clôture $\overline{L}$.

La preuve de ce résultat se fait par induction sur la longueur des suffixes composant l'ensemble L à reconnaître et en utilisant un « raisonnement par l'absurde » pour comparer les arbres construits par les algorithmes 5.1 et 5.2.

Il est clair que la clôture et la clôture réduite de l'ensemble $L = \{\epsilon\}$ sont égales : $\overline{\{\epsilon\}} = \widetilde{\{\epsilon\}} = \{\epsilon\}$.

Partant d'un automate canonique A et d'un automate faiblement canonique à mémoire B équivalent, nous voulons montrer que l'ajout d'un suffixe p, en utilisant les algorithmes 5.1 et 5.2, produit bien deux automates A' et B' équivalents.



(A') (B')

FIG. 5.5 – *Les deux automates A' et B' illustrent le résultat de l'insertion du suffixe $p = \alpha s t_1 \dots t_{\#s} \beta$ où s est un symbole tel qu'une arête étiquetée par s doit être créée. Dans le cas de A', la présence du suffixe $\alpha\omega\gamma$ (branche droite) provoque une insertion en « cascade » des suffixes $\omega^{\#s}\gamma$. Lors de la construction de la clôture réduite (B'), cet effet « boule de neige » est évité et seule la branche γ est dupliquée.*

Si l'on compare les deux algorithmes, on s'aperçoit qu'ils ne diffèrent qu'à deux endroits:
- dans la partie **si** $s \neq \omega$ **alors** ... **sinon**: lorsque l'arête étiquetée par s est créée, le deuxième algorithme ajoute moins de suffixes;
- dans la partie **sinon** ... **finsi**: lorsque le suffixe commence par un ω, le deuxième algorithme ajoute, ici aussi, moins de suffixes.

Soit p le suffixe à insérer, appelons s le premier symbole qui est un ω ou qui entraîne la création d'une arête. Le suffixe p est alors de la forme $p = \alpha s t_1 \dots t_{\#s} \beta$ où α et β sont des chaînes de Σ^*. La partie correspondant à α est insérée de la même manière par les deux algorithmes: les arêtes de l'automate existant sont suivies en fonction des symboles rencontrés. Lorsque le symbole s arrive en tête du suffixe à insérer, deux cas peuvent se produire:

1. $s \neq \omega$:
   Une arête étiquetée par s est créée, c'est donc que la branche correspondant à $st_1 \dots t_{\#s}\beta$ est nouvelle ($L/s = \emptyset$). La figure 5.5 représente de manière schématique la structure des arbres construits.
   Supposons qu'il existe un terme clos t qui soit reconnu par l'automate A' mais pas par l'automate à mémoire B'. Cela signifie que l'automate B' s'arrête dans un état non terminal. Cet état fait partie des nouveaux états construits pour passer de B à B', sinon le terme t aurait été reconnu par A et donc par B (A et B sont équivalents par hypothèse d'induction). Le terme t étant reconnu par A', cela signifie qu'il n'y a pas d'échec dans les branches β et γ, l'automate B' s'arrête donc dans un état de la branche $st_1 \dots t_{\#s}$. Mais, par définition, un automate à mémoire ne peut pas s'arrêter dans une branche correspondant à un terme qui aurait pu être complètement lu par une règle de transition étiquetée par un ω. Il y a donc une contradiction qui montre qu'il ne peut pas exister de terme t reconnu par A' qui ne soit pas reconnu par B'.

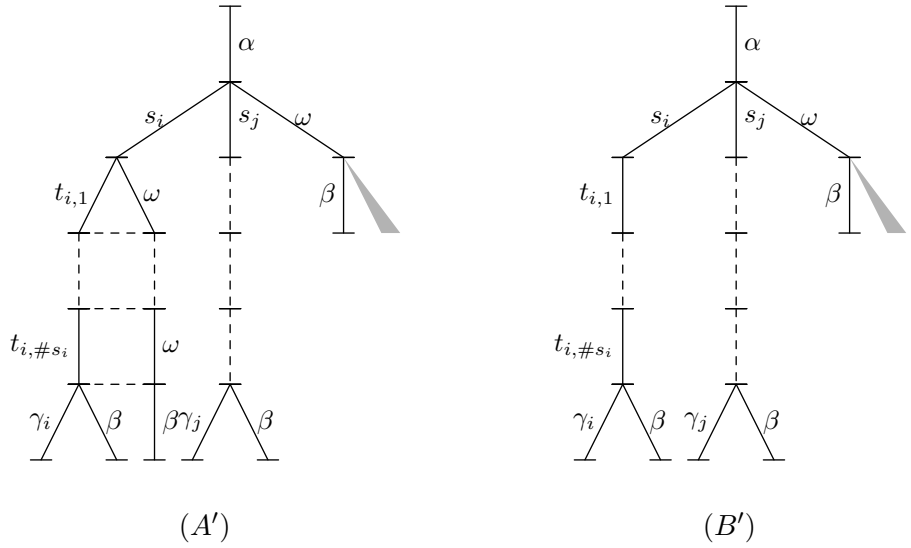


FIG. 5.6 – *Les deux automates A' et B' illustrent le résultat de l'insertion du suffixe $p = \alpha\omega\beta$. Le triangle grisé représente le sous-arbre déjà présent avant l'insertion de β. Le suffixe β est copié dans toutes les autres branches commençant par un symbole $s_i \neq \omega$. Il faut noter, dans le cas de A', que pour chaque sous-arbre $s_i \neq \omega$, un effet « boule de neige » se produit, dupliquant un très grand nombre de fois le suffixe β.*

2. $s = \omega$:
   En s'appuyant sur la figure 5.6, nous effectuons le même raisonnement que précédemment en supposant qu'il existe un terme clos t qui soit reconnu par l'automate A' mais pas par l'automate à mémoire B'. L'échec ne peut pas se produire dans les branches β et γ_i, ni dans la partie grisée, (sinon t ne pourrait être reconnu par A'), c'est donc qu'il se produit dans la partie correspondant au terme $s_i t_1 \dots t_{\#s_i}$ (pour un $s_i \neq \omega$). Comme il existe une arête ω partant du nœud d'où part l'arête s_i, cette situation ne peut pas se produire avec un automate de filtrage à mémoire. Nous obtenons une nouvelle contradiction qui prouve qu'il ne peut pas exister de terme t reconnu par A' qui ne soit pas reconnu par B'.

Cette étude par cas montre, qu'étant donné un ensemble L, les automates canoniques et faiblement canoniques à mémoire associés reconnaissent le même langage.

5.6 Automate de filtrage avec jumpNode

D'un point de vue théorique, l'automate faiblement canonique à mémoire est satisfaisant : son nombre d'états est suffisamment petit et il permet de reconnaître un langage sans inspecter plus d'une fois le terme d'entrée. D'un point de vue pratique, les résultats obtenus ne sont pas encore satisfaisants. Dans un logiciel permettant d'exécuter un système de réécriture, la procédure de filtrage est un composant crucial : elle est utilisée avant chaque application de règle pour déterminer quelles sont celles qui peuvent potentiellement s'appliquer. C'est pour cela que nous tenons à ce que les automates soient particulièrement optimisés.

Au cours du filtrage, lorsqu'un choix se présente, l'automate doit mémoriser l'état courant et tous les symboles qui vont être lus. Puis, lorsque l'automate se bloque dans un état, il doit inspecter sa mémoire pour y trouver le choix « responsable » de ce blocage. Dans cette partie,

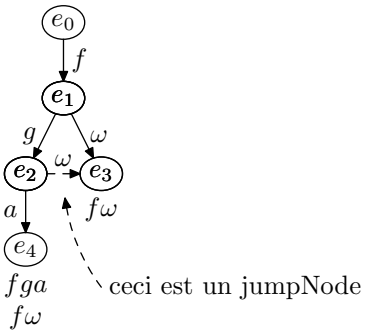


FIG. 5.7 – *Cet automate a été construit à partir d'une « clôture réduite », mais n'étant pas canonique, une règle de transition d'états ($\delta_i = (e_2,\omega) \longrightarrow_\delta e_3$) a été ajoutée afin de permettre un « changement de branche » et remettre en cause le dernier choix effectué. Il permet en particulier de reconnaître le terme fgb sans « bloquer » l'automate dans l'état e_2.*

nous proposons d'étendre l'automate à mémoire, en lui ajoutant des règles de transition d'états et une règle de transition de configuration, pour l'affranchir de ces deux étapes de mémorisation et de recherche. Commençons par faire quelques remarques :

- lorsque l'automate doit s'arrêter et faire une recherche, il est forcément dans un état d'où aucune arête étiquetée par un ω ne part ;
- si l'automate trouve, dans sa mémoire, un état e_1 (appelé « choix-ω ») d'où part une arête étiquetée par un ω ($\delta_i : (e_1,\omega) \longrightarrow_\delta e_2$) lui permettant, en utilisant le troisième mode de déplacement, d'atteindre une position plus à droite, cet état e_1 peut être déterminé statiquement en analysant la structure de l'automate : pour chaque état susceptible d'arrêter l'automate, il suffit de parcourir en sens inverse les arêtes jusqu'à trouver un embranchement avec une arête ω qui aurait permis d'atteindre un point plus à droite sur la bande de lecture. Si aucun choix-ω n'est trouvé, c'est qu'aucun état convenable n'aurait été trouvé dans la mémoire et l'automate peut potentiellement se bloquer ;
- étant donné un état e susceptible d'arrêter l'automate et son choix-ω ($\delta_i : (e_1,\omega) \longrightarrow_\delta e_2$) correspondant, le fait d'ajouter une règle de transition d'états $\delta_j : (e,\omega) \longrightarrow_\delta e_2$ appelée jumpNode (voir figure 5.7), permettant de passer de l'état bloquant e à l'état e_2, en utilisant le troisième mode de déplacement de la tête de lecture, simule le comportement de l'automate à mémoire.

L'automate de filtrage est une représentation d'un ensemble de suffixes L. Si, après avoir lu une suite de symboles $\alpha \in \Sigma^*$, un choix entre le symbole s et ω apparaît, c'est que les deux termes $\alpha s t_1 \dots t_{\#s}\beta$ et $\alpha\omega\beta'$ appartiennent à l'ensemble L. Pour tout état e, susceptible de bloquer l'automate, tel que e soit un état de la branche correspondant à $st_1 \dots t_{\#s}$ (en supposant qu'il n'y a pas de choix-ω dans cette branche), un jumpNode, reliant e à l'état fils de l'arête ω, doit être créé.

Partant d'un automate de filtrage correspondant à une « clôture réduite », la construction d'un automate de filtrage avec jumpNode se décompose en trois étapes :

1. si aucune arête étiquetée par le symbole ω ne part de l'état initial e_0, un état particulier d'échec e_j ainsi qu'une règle de transition d'états $\delta_i = (e_0,\omega) \longrightarrow_\delta e_j$ sont ajoutés à l'automate pour permettre de prendre en compte un échec du filtrage.

2. l'algorithme 5.3 est appliqué récursivement sur l'état initial de l'automate (en partant d'une pile vide) pour construire un chaînage qui associe à chaque état de l'automate un lien vers son « père ». La notion de « père » d'un état correspond à la notion de « père » définie sur les termes : considérons par exemple le terme $\alpha s t_1 \dots t_{\#s} \beta$ et son sous-terme t_i, le symbole s est appelé père de t_i parce que t_i est un sous-terme direct de s. Cette notion s'étend aux automates de filtrage de sorte que s'il existe deux règles de transition d'états $\delta_i = (e_i,s_i) \longrightarrow_\delta e'_i$ et $\delta_j = (e_j,s_j) \longrightarrow_\delta e'_j$ telles que s_i soit le père des sous-termes $s_j t_1 \dots t_{\#s_j}$, l'état e_i est appelé « père » de e_j. Il faut noter que l'état initial n'a pas de « père ».

Algorithme 5.3 Construction du lien vers le père

```
1: père(e1,pile1) =
2: pour tout δi : (e1,si) ⟶δ e2, (on peut avoir si = ω) faire
3:    pile2 ← pile1
4:    si #si > 0 alors
5:       empiler (e1,#si) dans pile2
6:    sinon si pile2 non vide alors
7:       arret ← ⊥
8:       (e,niveau) ← dépiler pile2
9:       chaîner e1 vers e
10:      tantque niveau = 1 et arret ≠ ⊤ faire
11:         si pile2 non vide alors
12:            (e',niveau') ← dépiler pile2
13:            chaîner e vers e'
14:            (e,niveau) ← (e',niveau')
15:         sinon
16:            arret ← ⊤
17:         finsi
18:      fin tantque
19:      si arret ≠ ⊤ alors
20:         empiler (e,niveau − 1) dans pile2
21:         appeler récursivement père(e2,pile2) sur les fils e2
22:      finsi
23:   finsi
24: fin pour
```

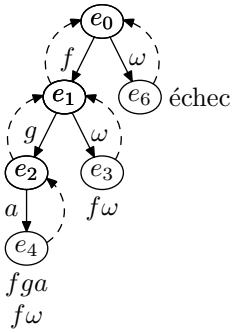
3. la construction des jumpNode s'effectue en appliquant l'algorithme 5.4 sur l'état initial de l'automate de filtrage enrichi par ce chaînage vers les « pères ».

Exemple 3 *Considérons une nouvelle fois l'automate de filtrage associé à la « clôture réduite » de l'ensemble de motifs $L = \{fga, f\omega\}$, l'ajout d'un état particulier d'échec suivi de l'application*

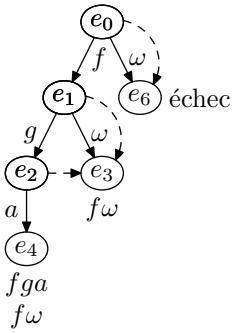
Algorithme 5.4 Construction des jumpNode

1: soit un état non final e_1
2: **si** il existe une règle de transition d'états δ_i : $(e_1,\omega) \longrightarrow_\delta e_2$ **alors**
3: jumpNode(e_1) $\leftarrow e_2$
4: **sinon**
5: jumpNode(e_1) $\leftarrow$ jumpNode(père(e_1))
6: **finsi**
7: l'algorithme est appliqué récursivement à tous les fils de e_1

de l'algorithme 5.3 sur l'état initial e_0 nous donne l'automate enrichi suivant :



Il suffit alors d'appliquer l'algorithme 5.4 sur l'état e_0 pour obtenir l'automate avec jumpNode suivant :



La partie intéressante de l'algorithme est appliquée lorsqu'il s'agit de calculer le jumpNode associé à l'état e_2 : aucune arête étiquetée par ω ne part de cet état, il faut donc utiliser le premier chaînage pour remonter à l'état père de e_2 (c'est l'état e_1). Il suffit alors d'utiliser le deuxième chaînage pour récupérer le jumpNode e_3 associé à e_1 et l'associer aussi à l'état e_2.

5.7 Comparaison des différentes approches

Dans ce chapitre, nous avons présenté un nouvel algorithme permettant de construire des automates de filtrages déterministes. Il faut avouer que le domaine n'est pas réellement nouveau, et pourtant l'algorithme présenté ne ressemble à aucun autre. La première originalité est qu'il permet d'obtenir des automates déterministes, alors que les travaux ont plus souvent porté sur l'étude des automates non-déterministes. Aucune des deux approches n'est meilleure que

l'autre, tout dépend de l'utilisation qu'on veut faire de l'automate. Dans le cadre d'un prouveur de théorèmes ou d'un système de règles qui évolue dynamiquement au cours du temps, il est souvent préférable d'avoir un algorithme permettant d'ajouter ou d'enlever efficacement des motifs de l'ensemble reconnu par l'automate. Les algorithmes de construction d'automates de filtrage non-déterministes possèdent habituellement ces bonnes propriétés. Par contre, les automates ainsi construits sont souvent moins performants que leurs versions déterministes. Dans le cadre d'un compilateur de systèmes de réécriture, il est préférable d'avoir des automates performants même si le temps de construction de ces automates est plus élevé que celui des versions non-déterministes.

En 1996, notre objectif n'était pas d'inventer un nouvel algorithme, mais d'en implanter un pour l'intégrer au compilateur ELAN. Après l'étude de trois présentations différentes (Gräf 1991, Sekar et al. 1992, Graf 1996), la version présentée dans la thèse de Peter Graf (1996) nous a semblé être la plus facile à implanter, simplement parce que les algorithmes permettaient de manipuler directement des automates sans avoir à calculer des clôtures d'ensembles de motifs au préalable. D'un point de vue implantation, il était préférable de ne manipuler qu'un seul type de données tel que les automates ou les arbres de filtrage. Il s'est avéré, après implantation, que les automates obtenus ne donnaient pas les résultats attendus : certains motifs n'étaient pas reconnus alors qu'ils filtraient effectivement le sujet. L'étude détaillée de l'algorithme et des problèmes rencontrés a permis de constater que les automates produits ne correspondaient qu'à des sous-ensembles des « clôtures réduites » présentées dans ce chapitre. C'est donc en essayant de rentabiliser notre premier investissement que nous avons corrigé l'algorithme et finalement développé un nouvel algorithme pour construire des automates déterministes.

Par rapport à l'algorithme d'Albert Gräf (1991), notre approche a l'avantage de proposer une version constructive et incrémentale de l'algorithme : les motifs sont insérés un à un dans une structure arborescente. Après chaque insertion, l'arbre obtenu est un automate directement exploitable. Il n'est pas nécessaire de calculer une clôture, puis d'en dériver un automate. Le deuxième avantage concerne la taille mémoire occupée par l'automate : l'utilisation de jumpNode et d'un automate correspondant à une « clôture réduite » permet de réduire considérablement le nombre d'états composant l'automate tout en améliorant ses performances.

En 1997, un nouvel algorithme présenté dans (Nedjah et al. 1997), a permis d'obtenir des automates de filtrage « optimaux » en terme de taille occupée. Ces automates ne peuvent cependant s'utiliser que pour filtrer des ensembles de motifs avec priorité. Lorsque plusieurs motifs filtrent un sujet, l'automate n'est pas capable de donner la liste de motifs : seul le motif ayant la plus grande priorité peut être donné. L'approche utilisée consiste à calculer dans un premier temps une clôture de l'ensemble de motifs. Une deuxième étape inspecte alors la clôture pour y rechercher des sous-ensembles « équivalents ». La recherche de ces sous-ensembles et la vérification de leur équivalence sont des opérations complexes qui permettent néanmoins de réduire la taille des automates générés en mettant en facteur certains ensembles d'états. Un exemple donné dans l'article (dans le cas où $L = \{f(a,a,x,a), f(g(a,x),a,a,b), f(x,b,b,b)\}$) montre que l'approche proposée permet de passer d'un automate possédant 27 états à un automate possédant seulement 15 états. Pour ce même exemple, il est intéressant de constater que notre approche utilisant des jumpNode permet de construire un automate possédant seulement 17 états. La comparaison des automates ainsi construits montre que les deux états supplémentaires peuvent être partagés, en modifiant l'algorithme de construction des automates de la manière suivante : si lors de l'insertion d'un terme dans l'arbre de filtrage un nœud doit être créé et que ce terme se trouve déjà à une autre position de l'arbre, un jumpNode peut être ajouté pour partager ce terme. Par la suite, lorsqu'un sous-arbre partagé est modifié par l'insertion d'un nouveau terme, il faut alors dupliquer la partie partagée pour éviter tout effet de bord. Il faut cependant noter que cette

optimisation (qui est à rapprocher de celle décrite dans (Nedjah et al. 1997)) n'a d'intérêt que si les feuilles de l'arbre considéré ne correspondent qu'à un seul motif. On s'aperçoit alors que l'exemple choisi n'est pas tellement représentatif de ce qui se passe en pratique et l'algorithme présenté dans (Nedjah et al. 1997) ne permet finalement pas un si bon partage lorsque l'ensemble de motifs contient des termes plus généraux que d'autres. Et c'est malheureusement ce qui se produit le plus souvent en pratique. Considérons une fois encore l'exemple de la fonction factorielle où les motifs impliqués sont $L = \{fact(0), fact(1), fact(x)\}$. Dans cet exemple, le terme $fact(x)$ est une généralisation des termes $fact(0)$ et $fact(1)$.

Remarquons que la méthode utilisant des jumpNode augmente légèrement l'efficacité du filtrage parce que le sujet n'a pas toujours besoin d'être parcouru entièrement pour déterminer les règles qui peuvent s'appliquer. Reprenons l'exemple donné précédemment et considérons le terme $f(g(b,a),b,b,b)$. Les automates déterministes construits par les algorithmes (Gräf 1991, Nedjah et al. 1997) inspectent toutes les positions du terme pour trouver que le motif $f(x,b,b,b)$ peut s'appliquer. En utilisant un automate avec jumpNode, après avoir lu les symboles f, g et b, un « saut » est effectué pour lire les trois symboles b. La position associée au symbole a n'a pas besoin d'être inspectée pour trouver que seul le motif $f(x,b,b,b)$ filtre le terme, d'où une meilleure efficacité de la procédure de filtrage. Notre approche, qui permet de traiter les ensembles de motifs sans priorité et qui n'est pas strictement *left-to-right* est donc à rapprocher des techniques de constructions d'automates adaptatifs décrites dans (Sekar et al. 1992). Mais à la différence des automates adaptatifs, qui modifient complètement l'ordre de parcours du sujet, notre algorithme garantit un parcours en profondeur d'abord des sous-termes du sujet, ce qui permet d'utiliser une structure aplatie de termes (*flatterms*) telle que celle décrite dans (Christian 1993).

Chapitre 6

Compilation du filtrage associatif-commutatif

6.1 Termes en forme canonique . . . 82
6.2 Approche one-to-one . . . 83
6.3 Approche many-to-one . . . 84
6.4 Classes de motifs . . . 88
6.5 Spécialisation utilisant une structure compacte . . . 88
6.6 Raffinement glouton . . . 93
6.7 Calcul des substitutions . . . 94
6.8 Extension à l'ensemble des motifs . . . 95
6.9 Synthèse . . . 97

À l'image du chapitre précédent, le problème traité dans ce chapitre consiste aussi à sélectionner une règle parmi un ensemble, pour réduire un terme clos donné. Le problème est néanmoins légèrement différent dans la mesure où les membres gauches des règles peuvent contenir des symboles associatifs et commutatifs. La principale difficulté introduite consiste alors à sélectionner une règle dont le membre gauche filtre le sujet modulo les axiomes d'associativité et de commutativité. Le problème n'est une fois de plus pas nouveau dans la mesure où il a été intensivement étudié dans (Hullot 1980, Benanav, Kapur et Narendran 1987, Kounalis et Lugiez 1991, Bachmair, Chen et Ramakrishnan 1993, Lugiez et Moysset 1994, Eker 1995, Moreau et Kirchner 1998).

Ici encore, nous nous intéressons aux algorithmes de filtrage *many-to-one*, mais à la différence de la théorie syntaxique, la résolution d'un problème de filtrage AC peut avoir plusieurs solutions. On imagine alors facilement qu'un algorithme permettant de calculer une solution du problème de filtrage correspondant à l'application d'une seule règle, n'a pas le même coût et ne s'utilise pas de la même façon qu'un algorithme retournant toutes les solutions associées aux problèmes de filtrage relatifs à l'ensemble des règles pouvant s'appliquer. C'est pourquoi il est important de clarifier le contexte d'utilisation afin de déterminer quel type d'algorithme doit être étudié.

Rappelons que notre objectif est de réaliser un compilateur permettant de calculer des formes normales de termes par rapport à un système de règles de réécriture conditionnelles. Pour cela nous avons besoin d'un algorithme de filtrage AC satisfaisant les conditions suivantes :

– Étant donnés un terme s et un ensemble de règles $\{p_1 \to r_1, \ldots, p_n \to r_n\}$, il doit sélectionner rapidement une règle de sorte qu'il existe une substitution σ telle que $p_i\sigma =_{AC} s$. Il doit ensuite trouver une telle substitution σ du problème de filtrage considéré et surtout permettre de construire efficacement une telle substitution.

– L'utilisation de règles conditionnelles fait que pour une solution σ donnée, les conditions peuvent ne pas être satisfaites, il faut donc que l'algorithme de filtrage soit capable d'extraire successivement toutes les solutions d'un problème de filtrage donné. Mais il n'est pas nécessaire de calculer cet ensemble de solutions en une seule fois.
– Lorsque pour une règle de réécriture donnée l'algorithme ne fournit pas de solution satisfaisant les conditions, il doit être capable de sélectionner efficacement une autre règle de réécriture. Il serait alors intéressant de récupérer une partie du travail effectué pendant les premières tentatives infructueuses. On peut remarquer qu'il n'est pas nécessaire de sélectionner l'ensemble des règles en une seule fois : celles-ci peuvent être déterminées une à une.

L'approche décrite dans (Bachmair et al. 1993) est sûrement celle qui se rapproche le plus de nos attentes, mais elle ne permet malheureusement pas de construire efficacement les solutions et les substitutions associées à un problème de filtrage AC. Notre objectif n'est pas de définir une nouvelle procédure générale de filtrage AC, mais l'étude de ces travaux nous a amenés à définir une procédure de filtrage qui permet de calculer efficacement des formes normales de terme clos en utilisant des règles de réécritures conditionnelles pouvant faire intervenir des symboles AC.

Les procédures de filtrage AC ont en général une complexité importante (Benanav et al. 1987, Hermann et Kolaitis 1995), qui est polynômiale lorsque les motifs considérés sont linéaires. L'étude empirique d'un grand nombre de systèmes de réécriture a montré que la majeure partie des motifs AC utilisés en pratique appartiennent finalement à une classe assez restreinte de termes. La définition et l'étude de ces classes de motifs nous ont permis de définir un algorithme de filtrage limité mais très efficace pour ces classes de motifs. Le cas général, qui se présente assez rarement, étant traité par l'algorithme présenté dans (Eker 1995).

La deuxième particularité de notre approche est relatif à l'aspect *compilation* largement abordé dans cette thèse. En effet, il ne s'agit pas seulement de définir un algorithme de filtrage « efficace », il faut aussi que celui-ci puisse facilement s'intégrer dans la réalisation d'un compilateur. Ce point, abordé dans le chapitre 4, implique qu'un grand nombre des structures de données de l'algorithme doivent se traduire en des structures de contrôle du langage cible choisi. Ce chapitre présente donc l'algorithme de filtrage AC développé dans le cadre de la réalisation du compilateur. Une des difficultés de cette présentation réside dans la distinction entre le travail effectué au cours de la compilation (c'est la phase de génération d'un algorithme de filtrage pour un problème donné), et celui fait au cours de l'exécution du code généré (c'est la phase qui sélectionne une règle et calcule un filtre s'il existe).

6.1 Termes en forme canonique

Lorsqu'on manipule des termes comportant des symboles AC, il est fréquent que les termes soient syntaxiquement différents mais tout en représentant les mêmes objets mathématiques. Nous avons vuau paragraphe 1.5 que les polynômes $(3 * X * X) + (2 * X) + 1$ et $(X * 2) + 1 + (X * 3 * X)$ ne s'écrivent pas de la même façon mais qu'ils sont bien égaux modulo les axiomes d'associativité et de commutativité. Dans une telle situation où l'on considère des termes qui sont dans une même classe d'équivalence, il est souvent pratique, voire indispensable de choisir une représentation « conventionnelle » appelée *représentation canonique* ou *forme canonique* pour décrire et appliquer les algorithmes.

Dans le cadre des théories AC, la notion de terme aplati est essentielle : si on oriente l'équation d'associativité en la règle $f_{AC}(f_{AC}(x,y),z) \rightarrow f_{AC}(x,f_{AC}(y,z))$, on obtient un système qui termine et le terme obtenu à partir d'un terme t s'appelle la forme aplatie de t. Un terme aplati

est un terme $f_{AC}(t_1,f_{AC}(t_2,\dots f_{AC}(t_{n-1},t_n)))$ tel que la racine d'aucun des t_i n'est f_{AC} et on le note $f_{AC}(t_1,\dots,t_n)$. Si on se donne un ordre total $<$ sur les symboles et son extension aux termes, la forme canonique d'un terme aplati t peut être obtenue en triant les sous-termes (eux-mêmes en forme canonique) et en remplaçant α sous-termes t identiques par une occurence de t unique avec multiplicité notée t^α. En se donnant un ordre total sur les entiers, les noms de variables et les symboles $*$ et $+$ ($0 < 1 < \dots < X < Y < * < +$), la forme canonique des deux polynômes présentés précédemment est $+(*(3,X^2),*(2,X),1)$ (nous avons utilisé une notation préfixée des opérateurs $*$ et $+$ pour plus de clarté). Il ne faut pas confondre la notation X^2 avec l'élévation à la puissance définie sur le corps des polynômes. En effet, la forme normale du terme $X+X$ se note $+(X^2)$ et n'a aucun rapport avec la notion habituelle de « X au carré ».

Un terme en forme canonique est dit *semi-linéaire* si le terme obtenu en « oubliant » les multiplicités des variables apparaissant sous un symbole AC est lui-même linéaire. Si x et y sont des variables, le terme $f_{AC}(x^3,y^2,g(a))$ est semi-linéraire mais pas les termes $f_{AC}(y,g(x)^2)$ et $f_{AC}(x,y^2,g(x))$.

6.2 Approche one-to-one

Pour pouvoir calculer la forme normale d'un terme par rapport à un système de réécriture comportant des symboles AC, l'approche la plus simple consiste à se donner un algorithme prenant en argument une règle de réécriture $p \to r$, un terme clos s et retournant l'ensemble des substitutions σ telles que $p\sigma$ et s soient égaux modulo AC. Un tel algorithme est connu sous le nom de *procédure de filtrage AC one-to-one*. La résolution des problèmes de filtrage AC, notés $p \leq^?_{AC} s$ est connue pour être NP-complet (Benanav et al. 1987, Hermann et Kolaitis 1995). Des méthodes de résolution, néanmoins efficaces en pratique, ont été proposées dans (Hullot 1979, Hullot 1980, Eker 1995) par exemple.

Dans ce paragraphe nous présentons les grandes lignes d'un tel algorithme afin d'introduire les concepts nécessaires à la bonne compréhension des méthodes proposées dans la suite du chapitre.

Étant donnés un motif p et un sujet s en forme canonique, la résolution de $p \leq^?_{AC} s$ nous amène à considérer les notions de *couche supérieure syntaxique* et de sous-problème de filtrage AC. La *couche supérieure syntaxique* d'un terme t en forme canonique se note $\hat{t}$ et correspond à l'élimination de tous les sous-termes de t apparaissant directement sous un symbole AC. Considérons le terme $t = expand(*(+(a,b),+(c,d)),n)$ où $+$, $*$ sont des opérateurs AC et a,b,c,d,n des variables, on a alors $\hat{t} = expand(*,n)$. Il faut noter que le terme $\hat{t}$ est bien formé si les symboles AC sont vus comme des constantes.

La première étape de la procédure de filtrage AC consiste à utiliser le filtrage syntaxique pour tester s'il existe un filtre de $\hat{p}$ vers $\hat{s}$ et savoir si $p \leq^?_{AC} s$ a potentiellement une solution. Dans l'affirmative, le problème $p \leq^?_{AC} s$ se décompose en autant de nouveaux sous-problèmes de filtrage qu'il y a de symboles AC dans $\hat{p}$. Ces problèmes sont de la forme :

$$f_{AC}(p_1^{\alpha_1},\dots,p_k^{\alpha_k},p_{k+1}^{\alpha_{k+1}},\dots,p_n^{\alpha_n}) \leq^?_{AC} f_{AC}(s_1^{\beta_1},\dots,s_m^{\beta_m})$$

où tous les $p_1,\dots,p_k$ sont des variables, et aucun des $p_{k+1},\dots,p_n$ n'est une variable.

Résoudre un de ces problèmes revient à considérer une structure de donnée $BG = (V_1 \cup V_2,E)$ appelée *graphe biparti*. Un tel graphe est composé de deux ensembles de sommets $V_1 = \{s_1^{\beta_1},\dots,s_m^{\beta_m}\}$, $V_2 = \{p_{k+1}^{\alpha_{k+1}},\dots,p_n^{\alpha_n}\}$ et d'un ensemble d'arêtes E qui sont les paires $[s_j,p_i]$ telles que $p_i \leq^?_{AC} s_j$ ait une solution. La construction d'un tel graphe se fait en appliquant

récursivement la procédure de filtrage AC, ce qui mène à la construction d'une *hiérarchie* de graphes bipartis.

Résoudre une telle hiérarchie revient à chercher séparément les solutions des graphes la composant et à vérifier la cohérence des solutions trouvées. Résoudre un graphe biparti $BG = (V_1 \cup V_2, E)$ consiste à trouver un couplage S tel que :

$$\begin{cases} S \subseteq E \\ card(\{[s_j, p_i] \in S \mid k+1 \leq i \leq n\}) = \alpha_i \\ card(\{[s_j, p_i] \in S \mid 1 \leq j \leq m\}) \leq \beta_j \end{cases}$$

Les méthodes les plus connues, parce que simples et efficaces, pour résoudre ce type de problème sont décrites dans (Hopcroft et Karp 1973, Fukuda et Matsui 1989) par exemple.

Une fois la hiérarchie de graphes bipartis résolue, il reste à calculer les instances des variables $p_1, \ldots, p_k$, ce qui revient à résoudre un problème de la forme :

$$f_{AC}(p_1^{\alpha_1}, \ldots, p_k^{\alpha_k}) \leq^?_{AC} f_{AC}(s_1^{\beta'_1}, \ldots, s_m^{\beta'_m})$$

où $\beta'_1, \ldots, \beta'_m$ sont de nouvelles multiplicités dépendant des solutions du graphe biparti précédent. Ce type de problème bien connu consiste à trouver des solutions entières non négatives du système d'équations diophantiennes :

$$\begin{array}{ccccccc} \alpha_1 X_{1,1} & + & \cdots & + & \alpha_k X_{k,1} & = & \beta'_1 \\ \vdots & & & & \vdots & & \vdots \\ \alpha_1 X_{1,m} & + & \cdots & + & \alpha_k X_{k,m} & = & \beta'_m \end{array}$$

et comme chaque variable $p_1, \ldots, p_k$ se voit assigner un ou plusieurs sous-termes, il faut aussi que pour tout $i \in \{1, \ldots, k\}$,

$$\Sigma_{j=1}^{m} X_{i,j} \geq 1$$

Les principales méthodes de résolution de systèmes d'équations diophantiennes sont décrites dans (Pottier 1990, Boudet, Contejean et Devie 1990, Domenjoud 1991, MacMahon 1916).

Reste alors à s'assurer que les solutions trouvées aux différents sous-problèmes $f_{AC}(p_1^{\alpha_1}, \ldots, p_n^{\alpha_n}) \leq^?_{AC} f_{AC}(s_1^{\beta_1}, \ldots, s_m^{\beta_m})$ sont bien cohérentes entre elles : en effet, une variable ne peut se voir assigner deux valeurs différentes. Afin de détecter le plus rapidement possible les échecs et obtenir une procédure de filtrage AC efficace, ces étapes de propagation et de vérification sont souvent intégrées aux processus de résolution des graphes bipartis et de résolution des systèmes d'équations diophantiennes.

Cette présentation schématique d'un algorithme de filtrage AC *one-to-one* montre que le processus est relativement complexe et coûteux. Il le devient encore plus lorsqu'il s'agit de déterminer, pour un ensemble de règles, quelles sont celles qui peuvent s'appliquer sur un terme clos s. Des études ont donc été menées pour essayer de mettre en facteur une partie du travail à effectuer : l'idée consiste à utiliser des structures arborescentes pour sélectionner plus efficacement les règles permettant de réduire le terme s. Ces méthodes s'appellent des *procédures de filtrage AC many-to-one*.

6.3 Approche many-to-one

Dans (Bachmair et al. 1993), un algorithme de filtrage AC fondé sur l'utilisation d'une *structure de filtrage AC* est présenté. Il ne permet que de traiter des ensembles de motifs linéaires

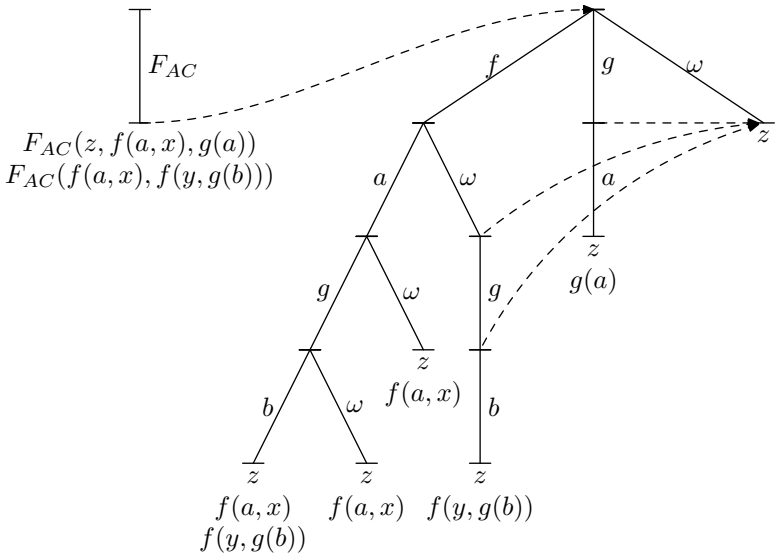


FIG. 6.1 – *Cette figure présente une structure de filtrage AC associée à l'ensemble de motifs* $P = \{f_{AC}(z,f(a,x),g(a)),f_{AC}(f(a,x),f(y,g(b)))\}$. *La partie droite de la figure illustre la réutilisation d'un automate de filtrage avec* jumpNode *associé à l'ensemble des sous-motifs* $\{z,f(a,x),g(a),f(y,g(b))\}$. *Cet automate est un composant de la structure de filtrage AC et son utilisation permet de construire efficacement les graphes bipartis associés à un problème de filtrage donné.*

mais le point particulièrement intéressant de cette approche est qu'elle permet de réutiliser une grande partie des travaux effectués sur les automates de filtrage, tels que ceux présentés dans le chapitre précédent. Un tel algorithme est dit *many-to-one* parce qu'étant donné un ensemble $P = \{p_1, \ldots, p_n\}$ et un terme clos s, il retourne l'ensemble $\{p_i \in P \mid p_i \leq^?_{AC} s\}$.

L'idée consiste à construire une collection d'automates de filtrage syntaxique en fonction de la structure des motifs de P. Ces différents automates sont ensuite « organisés » pour constituer une structure de filtrage AC. Partant d'un ensemble de motifs $P_i = P$, la construction d'une telle structure se décompose en quatre étapes :

1. calcul de la couche syntaxique supérieure $\hat{P}_i = \{\hat{p_{i_1}}, \ldots, \hat{p_{i_n}}\}$;
2. construction de l'automate de filtrage A_i associé à $\hat{P}_i$ où tous les symboles AC sont considérés comme des constantes ;
3. application récursive de l'algorithme à P_{i+1}, l'ensemble des termes « éliminés » au cours du calcul de $\hat{P}_i$;
4. construction d'un lien particulier entre les symboles AC de l'automate A_i et l'automate supérieur A_{i+1} de la sous-structure de filtrage AC construite lors de l'application récursive de l'algorithme.

Considérons par exemple l'ensemble de motifs $P_1 = \{f_{AC}(z,f(a,x),g(a)),f_{AC}(f(a,x),f(y,g(b))\}$ où seul f_{AC} est un symbole AC. On a $\hat{P}_1 = \{f_{AC},f_{AC}\}$ et l'ensemble des termes éliminés est $P_2 = \{z,f(a,x),g(a),f(y,g(b))\}$. Cette décomposition nous permet de construire la structure de filtrage AC illustrée dans la figure 6.1.

Tout comme la construction d'un automate de filtrage, les étapes précédentes dépendent seulement du système de réécriture et ne sont effectuées qu'une fois lors de la phase de compi-

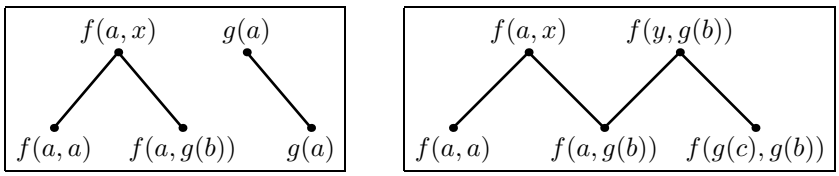


FIG. 6.2 – *Exemples de graphes bipartis associés aux problèmes de filtrage AC de* $f_{AC}(z,f(a,x),g(a))$ *et* $f_{AC}(f(a,x),f(y,g(b)))$ *vers* $f_{AC}(f(a,a),f(a,g(b)),\ f(g(c),g(b)),\ g(a))$

lation du système. Mais il faut bien noter que la structure de filtrage AC obtenue n'est pas un automate au sens strict : il ne suffit pas de parcourir le terme d'entrée et d'appliquer des règles de transition d'états pour obtenir l'ensemble des motifs qui filtrent le terme. La complexité même des problèmes de filtrage AC et la possibilité d'avoir plusieurs solutions pour un motif donné font que l'utilisation d'une structure de filtrage AC n'est qu'un moyen d'engendrer de nouveaux sous problèmes à résoudre.

Étudions maintenant comment utiliser la structure de filtrage AC pour sélectionner l'ensemble des motifs $\{p_i \in P \mid p_i \leq^?_{AC} s\}$. L'automate supérieur de la structure de filtrage AC est appliqué à la couche syntaxique supérieure du sujet $\hat{s}$ pour effectuer un pré-filtrage qui détermine l'ensemble des motifs restant candidats. Les sous-structures de filtrage AC associées aux symboles AC, rencontrés lors de l'application du premier automate sont alors utilisées pour vérifier que les sous-termes des motifs sélectionnés filtrent (modulo AC) les sous-termes du sujet. Au cours de l'application récursive de l'algorithme des graphes bipartis sont engendrés pour mémoriser les résultats intermédiaires.

Dans le cadre de notre exemple, on considère maintenant le terme clos (en forme canonique) $s = f_{AC}(f(a,a),f(a,g(b)),\ f(g(c),g(b)),\ g(a))$. Le premier automate de filtrage présenté sur la partie gauche de la figure 6.1 s'assure que le sujet commence bien par le symbole f_{AC} et indique que les motifs $f_{AC}(z,f(a,x),g(a))$ et $f_{AC}(f(a,x),f(y,g(b)))$ restent candidats pour la suite de l'étape de filtrage AC. L'application, sur les sous-termes $f(a,a)$, $f(a,g(b))$, $f(g(c),\ g(b))$ et $g(a)$, du deuxième automate de filtrage de la figure 6.1 nous amène alors à considérer les deux graphes bipartis donnés sur la figure 6.2 (un pour chaque règle). On peut noter qu'aucun sommet du graphe biparti de gauche ne correspond à la variable z. En effet, on sait à l'avance que la variable z filtre tous les sous-termes du sujet, il est donc inutile d'insérer dans le graphe une multitude d'arêtes qui ne peuvent qu'alourdir le processus de résolution. D'une manière générale, le calcul des instances de ces variables est pris en compte par une phase ultérieure de l'algorithme de filtrage décrite au paragraphe 6.7.

Dans (Bachmair et al. 1993), une méthode originale fondée sur l'utilisation d'automates est proposée pour tester l'existence d'une solution d'un graphe biparti donné : l'idée consiste à préconstruire des automates changeant d'état en fonction des configurations d'arêtes et indiquant à chaque instant si le graphe biparti considéré a au moins une solution. Le revers de la méthode est que le nombre d'états γ est exponentiellement proportionnel au nombre de sommets correspondant à des motifs (pour $V_2 = \{p_{k+1}^{\alpha_{k+1}}, \ldots, p_n^{\alpha_n}\}$, $\gamma = n - k$). Cela limite son application à des valeurs relativement petites de γ ($\gamma < 5$ par exemple), mais s'avère quand même suffisant en pratique : il est en effet assez rare de définir des systèmes de réécriture comportant des motifs dont le nombre de sous-termes d'un symbole AC soit grand. Lorsque cela se produit, une méthode plus générale, telle que celles décrites dans (Hopcroft et Karp 1973, Fukuda et Matsui 1989) peut alors être utilisée.

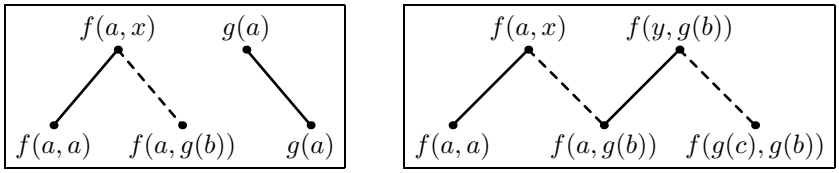


FIG. 6.3 – *Ces deux graphes représentent les solutions* $S_1 = \{[f(a,a),f(a,x)],[g(a),g(a)]\}$ *et* $S_2 = \{[f(a,a),f(a,x)],[f(a,g(b)),f(y,g(b))]\}$ *des deux graphes bipartis présentés sur la figure 6.2.*

Comme le montre la figure 6.3, les deux graphes bipartis de la figure 6.2 ont au moins une solution, il reste alors à vérifier pour un sous-problème de filtrage donné que :

- le nombre de variables apparaissant directement sous le symbole AC du motif est bien inférieur ou égal au nombre de sous-termes non impliqués dans une solution des graphes bipartis ;
- il y a au moins une variable apparaissant directement sous le symbole AC du motif si le nombre de sous-termes non impliqués dans une solution des graphes bipartis est non nul.

Une fois ces vérifications effectuées, l'étape de filtrage AC considérée peut retourner l'ensemble des motifs qui filtrent le sujet : les motifs vérifiant les deux critères précédents et dont le graphe biparti associé a au moins une solution.

Pour un ensemble de motifs P donné, l'approche décrite dans (Bachmair et al. 1993) est intéressante parce qu'elle permet de déterminer les motifs de P qui filtrent modulo AC un terme clos s en un temps $O(n) + O(mn^{1.5})$ où n est la taille du sujet s et m la somme des tailles des motifs de P. Utiliser une telle méthode pour réaliser un compilateur risquerait cependant de ne pas mener à l'implantation la plus efficace. C'est pourquoi dans la suite de ce chapitre, nous proposons un ensemble de spécialisations qui permettent d'améliorer l'efficacité de la procédure de filtrage, pour faire de la normalisation modulo AC :

- dans le cadre du calcul de la forme normale d'un terme par rapport à un système de réécriture il n'est pas nécessaire de connaître l'ensemble des règles pouvant s'appliquer sur un sujet, il suffit d'en sélectionner une seule. Cette remarque nous amène à résoudre successivement, et non plus simultanément, l'ensemble des graphes bipartis engendrés ;
- d'un point de vue implantation, la construction d'une hiérarchie de graphes bipartis est une opération coûteuse, dans la mesure où de nombreuses allocations dynamiques de mémoire doivent être effectuées. Nous proposons donc de limiter l'application de l'algorithme à une certaine classe de motifs, ce qui permet d'éviter la construction récursive d'une telle hiérarchie ;
- l'algorithme décrit dans ce paragraphe amène à construire autant de graphes bipartis que de motifs concernés par le problème de filtrage. Pour les même raisons d'efficacité que précédemment, nous proposons une nouvelle structure de *graphes bipartis compacts* permettant de représenter cet ensemble de graphes bipartis par une structure unique, limitant ainsi le nombre d'allocations dynamiques ;
- la présence de règles de réécriture conditionnelles amène à calculer les instances des variables impliquées pour déterminer si les conditions sont satisfaites. Lorsqu'elles ne le sont pas, il faut pouvoir extraire les autres solutions du problème de filtrage AC considéré. Nous proposons une méthode de compilation qui permet de calculer et de construire efficacement de telles instances de variables.

6.4 Classes de motifs

Après analyse d'un grand nombre de systèmes de réécriture et de spécifications écrites en ELAN, nous nous sommes aperçu que les membres gauches des règles utilisées suivaient souvent une certaine régularité. L'analyse fine, d'autre part, des algorithmes de filtrage AC, tel que celui présenté au paragraphe 6.3, nous a amené à isoler les étapes les plus complexes et les plus coûteuses. Partant de ces deux constats, nous avons défini des classes de termes représentant la majorité des motifs rencontrés mais permettant aussi d'affiner l'algorithme général de filtrage AC pour y éliminer les étapes les plus coûteuses, telles que la construction et la résolution des hiérarchies de graphes bipartis, ou encore la résolution des systèmes d'équations diophantiennes.

Les classes de motifs C_0,C_1 et C_2 contiennent respectivement les termes avec zéro, un ou deux niveaux de symboles AC. Soit $\mathcal{F}_\emptyset$ un ensemble de symboles de fonctions syntaxiques, $\mathcal{F}_{AC}$ un ensemble de symboles de fonctions AC et $\mathcal{X}$ un ensemble de variables, les classes de motifs se définissent de la manière suivante :

- la classe de motifs C_0 contient les termes linéaires $t \in \mathcal{T}(\mathcal{F}_\emptyset,\mathcal{X})\backslash\mathcal{X}$.
- la classe de motifs C_1 est le plus petit ensemble de termes semi-linéaires en forme canonique qui contient C_0 et tous les termes t de la forme :
  - $t = f_{AC}(x_1,x_2^{\alpha_2},t_1,\ldots,t_n)$, avec $f_{AC} \in \mathcal{F}_{AC}$, $0 \leq n$, $t_1,\ldots,t_n \in C_0$, $x_1,x_2 \in \mathcal{X}$, $\alpha_2 \geq 0$;
  - $t = f(t_1,\ldots,t_n)$, avec $f \in \mathcal{F}_\emptyset$, $t_1,\ldots,t_n \in C_1 \cup \mathcal{X}$.
- la classe de motifs C_2 est le plus petit ensemble de termes semi-linéaires en forme canonique qui contient C_1 et tous les termes t de la forme :
  - $t = f_{AC}(x_1,x_2^{\alpha_2},g_{AC}(x_3,x_4^{\alpha_4}))$, avec $f_{AC},g_{AC} \in \mathcal{F}_{AC}$, $x_1,x_2,x_3,x_4 \in \mathcal{X}$, $\alpha_2 \geq 0$, $\alpha_4 > 0$;
  - $t = f(t_1,\ldots,t_n)$, avec $f \in \mathcal{F}_\emptyset$, $t_1,\ldots,t_n \in C_2 \cup \mathcal{X}$.

On peut noter ici, qu'il est fréquent d'ajouter des *variables d'extension* aux membres gauche des règles pour assurer la complétude d'un système de réécriture modulo AC (Peterson et Stickel 1981, Jouannaud et Kirchner 1986). Ces variables d'extension permettent d'effectuer des réécritures sur les sous-termes en mémorisant le contexte d'application de la règle. L'ajout de telles variables nous amène à considérer les motifs de la forme $f_{AC}(x,t_1,\ldots,t_n)$ pour chaque règle dont le membre gauche est de la forme $f_{AC}(t_1,\ldots,t_n)$.

Dans notre exemple, les motifs $f_{AC}(z,f(a,x),g(a))$ et $f_{AC}(f(a,x),f(y,g(b)))$ ainsi que leur forme étendue $f_{AC}(z',z,f(a,x),g(a))$ et $f_{AC}(z,f(a,x),f(y,g(b)))$ appartiennent tous les quatre à la classe C_1.

6.5 Spécialisation utilisant une structure compacte

Partant de l'algorithme général présenté au paragraphe 6.3, nous proposons une nouvelle méthode de filtrage AC optimisée pour les classes de motifs définies au paragraphe 6.4. Les points clés de cette nouvelle approche sont les suivants :

- grâce aux restrictions faites sur les motifs, la structure de filtrage AC et la hiérarchie de graphes bipartis possèdent au plus deux niveaux et le deuxième niveau est dégénéré (*i.e.* de la forme $g_{AC}(x_3,x_4^{\alpha_4})$). La construction peut ainsi être faite sans récursivité ;
- nous utilisons une nouvelle représentation compacte des graphes bipartis qui permet de coder, dans une structure de donnée unique, l'ensemble des graphes bipartis relatifs au système de réécriture considéré ;

– il n'est plus nécessaire de construire et de résoudre des systèmes d'équations diophantiennes dans la mesure où il n'y a pas plus de deux variables sous un même symbole AC : l'instanciation de ces variables peut se faire en utilisant des méthodes simples et efficaces ;
– une analyse statique du système de réécriture permet de déterminer à l'avance les règles pour lesquelles il est suffisant de trouver une seule substitution. C'est le cas des règles sans condition ou des règles dont les conditions ne dépendent pas de variables apparaissant sous un symbole AC du membre gauche. Pour ces cas particuliers (mais fréquents), nous pouvons tirer parti de la structure de graphe biparti compact pour proposer un raffinement de l'algorithme de filtrage.

À l'image de l'algorithme général, la structure de filtrage AC est utilisée pour déterminer des couples de motifs et de termes clos, mais l'originalité de ce nouvel algorithme est d'exploiter au maximum les automates de filtrage syntaxique et d'éviter la construction d'une multitude de graphes bipartis. Considérons un sujet $s = f_{AC}(s_1^{\alpha_1}, \ldots, s_p^{\alpha_p})$ donné et un ensemble de motifs $p_1, \ldots, p_n$ de la forme :

$$\begin{array}{lcccc} p_1 & = f_{AC}(& p_{1,1} & \ldots, & p_{1,m_1} \quad) \\ \vdots & & \vdots & & \vdots \\ p_n & = f_{AC}(& p_{n,1} & \ldots, & p_{n,m_n} \quad) \end{array}$$

où pour k_j tel que $0 \leq k_j \leq m_j$, tous les $p_{j,1}, \ldots, p_{j,k_j}$ sont des variables et aucun des $p_{j,k_j+1}, \ldots, p_{j,m_j}$ n'est une variable.

Plutôt que de construire un graphe biparti BG_i associé à chaque motif p_i, ce qui peut obliger à filtrer n fois les sous-termes du sujet pour éviter la construction simultanée des graphes $BG_1, \ldots, BG_n$ nous construisons un graphe biparti compact unique qui contient les informations suffisantes pour pouvoir reconstruire n'importe quel BG_i. L'idée consiste à regrouper les $p_{j,k}$ et à définir le graphe biparti compact $CBG = (V_1 \cup V_2, E)$ dont les sommets sont $V_1 = \{\hat{s}_1^{\alpha_1}, \ldots, \hat{s}_p^{\alpha_p}\}$, $V_2 = \{\hat{p}_{j,k} \mid 1 \leq j \leq n, k_j + 1 \leq k \leq m_j\}$ et dont les arêtes E sont les paires $[\hat{s}_i, \hat{p}_{j,k}]$ telles que $\hat{p}_{j,k}$ filtre le terme clos $\hat{s}_i$.

Pour donner une meilleure intuition du processus, nous avons considéré un ensemble de motifs $p_1, \ldots, p_n$ ayant une même couche syntaxique supérieure réduite à f_{AC}, mais le nombre de symboles composant les couches syntaxiques supérieures n'a aucune importance. Il est par contre important de noter que les sommets du graphe biparti compact se composent de l'ensemble des motifs $\hat{p}_{j,k}$ apparaissant sous un symbole AC et que tous ces motifs sont des termes syntaxiques. Supposons que le motif p_1 appartienne à la classe C_2 et qu'il soit de la forme $f_{AC}(x_1, x_2^{\alpha_2}, g_{AC}(x_3, x_4^{\alpha_4}))$. On a alors $\hat{p}_{1,3} = g_{AC}$ qui est considéré comme un terme syntaxique, le calcul des instances des variables x_1, x_2, x_3 et x_4 étant fait dans une phase ultérieure de l'algorithme décrite au paragraphe 6.7.

Ces remarques étant faites, il est maintenant possible de percevoir les avantages apportés par l'utilisation des graphes bipartis compacts :

– leur construction se fait en utilisant uniquement des automates de filtrage syntaxique, puisque l'appel récursif de la procédure de filtrage AC n'est plus nécessaire ;
– les sous-termes $\hat{s}_1, \ldots, \hat{s}_p$ ne sont filtrés qu'une seule fois pour déterminer l'ensemble des arêtes E : l'automate de filtrage (qui est *many-to-one*) est appliqué sur chaque $\hat{s}_i$ pour déterminer l'ensemble des $\hat{p}_{j,k}$ qui filtrent $\hat{s}_i$.

Il reste maintenant à savoir comment reconstruire les graphes bipartis BG_j associés aux motifs p_j. Pour cela, il suffit de remarquer que les sommets qui composent BG_j constituent un

sous-ensemble des sommets composant CBG et que BG_j se calcule de la manière suivante :

$$BG_j = (V_1 \cup V_2', E') \text{ où } \left\{ \begin{array}{lcl} V_2' & = & \{\hat{p}_{j,k} \mid \hat{p}_{j,k} \in V_2 \text{ et } k_j + 1 \leq k \leq m_j\} \\ E' & = & \{[\hat{s}_i, \hat{p}_{j,k}] \mid [\hat{s}_i, \hat{p}_{j,k}] \in E \text{ et } \hat{p}_{j,k} \in V_2'\} \end{array} \right.$$

D'un point de vue implantation, cette extraction peut s'effectuer efficacement si l'on représente un graphe biparti compact par une structure de donnée adéquate. Le chapitre 11 présente une implantation à base de vecteurs qui ramène l'extraction d'un graphe biparti à l'extraction d'un ensemble de vecteurs.

Illustrons la méthode en l'appliquant sur l'exemple :

$$\begin{array}{lcll} f_{AC}(z, f(a,x), g(a)) & \rightarrow & r_1 & \textbf{if } z = x \\ f_{AC}(z', z, f(a,x), g(a)) & \rightarrow & f_{AC}(z', r_1) & \textbf{if } z = x \\ f_{AC}(f(a,x), f(y, g(b))) & \rightarrow & r_2 & \\ f_{AC}(z, f(a,x), f(y, g(b))) & \rightarrow & f_{AC}(z, r_2) & \end{array}$$

Le système considéré contient les deux règles de réécriture et leurs extensions respectives qui permettent leur application sur des sous-termes du sujet. La valeur des membres droits r_1 et r_2 n'est pas significative dans la mesure où nous nous intéressons à l'aspect filtrage AC du processus de normalisation. Il faut cependant noter que les deux premières règles comportent une condition booléenne **if** $z = x$ impliquant des variables du membre gauche : c'est ce type de situation qui peut amener l'algorithme de filtrage AC à devoir extraire plusieurs solutions pour en trouver une qui satisfasse la condition.

Appelons respectivement p_1, p_1' et p_2, p_2' les motifs des deux premières et des deux dernières règles, nous avons alors $p_{1,2} = p_{2,1} = f(a,x)$, $p_{1,3} = g(a)$, $p_{2,2} = f(y, g(b))$ et $p'_{j,k+1} = p_{j,k}$ (voir programme 6.1). Il faut noter que les sous-motifs composant une règle et son extension sont identiques. C'est pourquoi on peut oublier les sous-termes $p'_{j,k}$ dans la suite des explications.

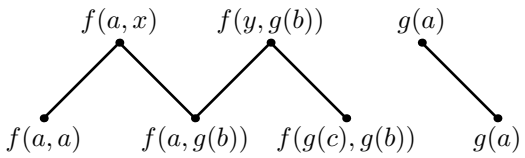
Initialisation des listes de motifs

```
void init_pattern_list_F() {
  /* F(z,zExt,f(a,x),g(a)) */
  pattern_tab[0]=0; pattern_tab[1]=1;
  MS_pattern_list_init(pattern_list_F,pattern_tab);
  /* F(zExt,f(y,g(b)),f(a,x)) */
  pattern_tab[0]=2; pattern_tab[1]=0;
  MS_pattern_list_init(pattern_list_F,pattern_tab);
}
```

Programme 6.1: *Cette fonction* `C` *est un exemple de programme qu'il est possible de générer à partir des algorithmes décrits dans ce chapitre. L'étude des exemples de code peut être évitée en première lecture. La fonction* `init_pattern_list_F`*, présenté ci-dessus, est exécutée au lancement du programme pour initialiser la construction des graphes bipartis compacts en donnant un numéro à chaque motif et à chaque sous-motif.* $f_{AC}(z, z', f(a,x), g(a))$ *et* $f_{AC}(z', f(y,g(b)), f(a,x))$ *deviennent les motifs* 0 *et* 1 *(*f_{AC} *est renommé en* `F` *et seules les règles ayant une variable d'extension sont conservées). Les numéros* 0, 1 *et* 2 *sont affectés, respectivement, aux sous-motifs* $f(a,x)$, $g(a)$ *et* $f(y,g(b))$.

En appliquant successivement l'automate de filtrage de la figure 6.1 (voir aussi programme 6.2) sur les sous-termes de $s = f_{AC}(f(a,a),f(a,g(b)),\ f(g(c),g(b)),\ g(a))$, on obtient les paires : $[f(a,a),p_{1,2} = p_{2,1}]$, $[f(a,g(b)),p_{1,2} = p_{2,1}]$, $[f(a,g(b)),p_{2,2}]$, $[f(g(c),g(b)),p_{2,2}]$ et $[g(a),p_{1,3}]$.

Ces paires sont utilisées pour construire le graphe biparti compact suivant :



Ce graphe biparti compact est ensuite utilisée pour normaliser le sujet s : une règle est sélectionnée, par exemple $f_{AC}(z,f(a,x),g(a)) \to r_1$ **if** $z = x$. Le graphe biparti BG_1 qui aurait été construit en appliquant la méthode générale s'obtient en sélectionnant les arêtes reliant les sommets $f(a,x)$ et $g(a)$ (voir partie gauche de la figure 6.2, page 86). La résolution de ce graphe biparti nous donne deux solutions :

$$\begin{array}{ll} S_1 = & \{[f(a,a),f(a,x)], \quad [g(a),g(a)]\} \\ S_2 = & \{[f(a,g(b)),f(a,x)], \quad [g(a),g(a)]\} \end{array}$$

Ces solutions permettent de calculer les différentes instances possibles de la variable x : $x \mapsto a$ ou $x \mapsto g(b)$.

Il reste alors à calculer, pour chaque motif, les instances des variables qui n'apparaissent pas dans les graphes bipartis (ce sont les variables qui sont directement sous un symbole AC). Dans notre exemple, il s'agit des variables z et z'. En effet, pour optimiser cette étape de résolution, on considère dans un même temps la règle et son extension : lorsque a est affecté à x, z prend pour valeur un sous-ensemble des sous-termes du sujet qui ne font pas partie de la solution du graphe biparti, et le complément est affecté à z', ce qui nous donne trois possibilités :

$$\begin{array}{l|l} z \mapsto f_{AC}(f(a,g(b)),f(g(c),g(b))) & z' \mapsto \emptyset \\ z \mapsto f(a,g(b)) & z' \mapsto f(g(c),g(b)) \\ z \mapsto f(g(c),g(b)) & z' \mapsto f(a,g(b)) \end{array}$$

et lorsque $x \mapsto f(a,g(b))$, on a :

$$\begin{array}{l|l} z \mapsto f_{AC}(f(a,a),f(g(c),g(b))) & z' \mapsto \emptyset \\ z \mapsto f(a,a) & z' \mapsto f(g(c),g(b)) \\ z \mapsto f(g(c),g(b)) & z' \mapsto f(a,a) \end{array}$$

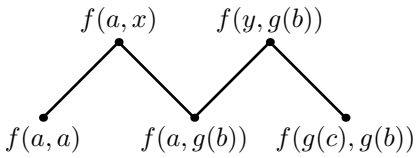
En aucun cas, la condition $z = x$ ne peut être satisfaite, ce qui conduit à un échec de l'application des deux premières règles.

Il faut donc sélectionner un autre ensemble de règles pouvant potentiellement s'appliquer :

$$f_{AC}(f(a,x),f(y,g(b))) \to r_2 \text{ et son extension } f_{AC}(z,f(a,x),f(y,g(b))) \to r_2$$

Il faut cette fois construire le graphe biparti BG_2 obtenu en extrayant les arêtes reliant les sommets $f(a,x)$ et $f(y,g(b))$. Il faut noter qu'aucune étape de filtrage supplémentaire n'est nécessaire pour construire ce nouveau graphe biparti : le travail est fait une seule fois lors de la construction du CBG :

$BG_2 =$



Compilation d'un automate de filtrage déterministe

```
int match_subterm_F(struct term *subject, int *mask) {
  switch(getSymb(subject)) {
  case code_g: successor_g=subject->subterm[0];
    switch(getSymb(successor_g)) {
    case code_a:
      mask[nb_bit++]=1;
      break;
    }
    break;
  case code_f: successor_f=subject->subterm[0];
    switch(getSymb(successor_f)) {
    case code_a: successor_a=subject->subterm[1];
      switch(getSymb(successor_a)) {
      case code_g: successor_g=successor_a->subterm[0];
        switch(getSymb(successor_g)) {
        case code_b:
          mask[nb_bit++]=0;
          mask[nb_bit++]=2;
          break;
        default: goto label7;
        }
        break;
      default:
      label7:
        mask[nb_bit++]=0;
      }
 ...
  }
  return nb_bit;
}
```

Programme 6.2: *Cette fonction implante l'automate de filtrage déterministe présenté dans la figure 6.1. Elle prend un terme clos* `subject` *en argument et parcourt les constructeurs qui le composent (*`code_f`*,* `code_g` *ou* `code_a` *par exemple). Lorsqu'un état final est atteint, les numéros des sous-motifs qui filtrent le terme clos sont mémorisés dans un tableau :* `mask`*. Ce tableau est ensuite utilisé pour construire le graphe biparti compact.*

La résolution de BG_2 nous amène à considérer trois solutions :

$$\begin{array}{lll} S_1 = & \{[f(a,a),f(a,x)], & [f(a,g(b)),f(y,g(b))]\} \\ S_2 = & \{[f(a,a),f(a,x)], & [f(g(c),g(b)),f(y,g(b))]\} \\ S_3 = & \{[f(a,g(b)),f(a,x)], & [f(g(c),g(b)),f(y,g(b))]\} \end{array}$$

Les deux règles considérées n'étant pas conditionnelles, l'une des deux pourra s'appliquer pour réduire le terme s.

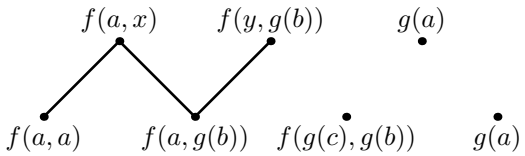
6.6 Raffinement glouton

Les problèmes de filtrage AC ont généralement plusieurs solutions, mais pour appliquer une règle de réécriture, il est souvent nécessaire de n'en calculer qu'une. En particulier, lorsqu'on considère des règles non conditionnelles ou des règles dont les conditions ne dépendent pas de variables apparaissant sous un symbole AC du membre gauche, le calcul d'une seule solution du problème de filtrage est suffisant pour appliquer la règle. Cette remarque nous a amené à définir une spécialisation de notre algorithme de filtrage pour ces règles dites *gloutonnes*. L'idée consiste à construire le graphe biparti compact de manière incrémentale et à ajouter une phase de vérification entre deux étapes : à chaque fois qu'un sous terme s_i du sujet est filtré, des arêtes sont ajoutées vers les motifs $p_{j,k}$ qui le filtrent. Si ces motifs apparaissent dans des membres gauches p_j de règles gloutonnes, un test d'existence de solution est appliqué aux BG_j correspondants, pour en extraire une solution. Lorsqu'une solution est trouvée, le processus de filtrage peut s'arrêter et retourner la solution. Lorsque tous les tests intermédiaires de statisfaisabilité échouent, la construction du graphe biparti compact se termine normalement. Il reste alors à extraire les graphes bipartis correspondant aux règles non gloutonnes, comme décrit au paragraphe 6.5.

Pour construire le graphe biparti compact de l'exemple traité au paragraphe 6.5, quatre étapes de filtrage étaient nécessaires, avant de commencer l'extraction des BG_j. En supposant que les sous-termes du sujet soient filtrés de la gauche vers la droite, l'application du raffinement glouton entraîne le filtrage de seulement deux termes pour produire la première solution :

$$S = \{[f(a,a),f(a,x)],[f(a,g(b)),f(y,g(b))]\}.$$

Il suffit de filtrer les termes $f(a,a)$ et $f(a,g(b))$ pour trouver un graphe biparti ayant une solution. Celle-ci est trouvée dès que le graphe biparti compact suivant est partiellement construit :



Le raffinement glouton a pour inconvénient d'introduire des étapes de vérification supplémentaires, mais en contre-partie, il permet de réduire considérablement le nombre de tentatives de filtrage. Dans l'algorithme du paragraphe 6.5, le nombre d'étapes de filtrage est égal au nombre de sous-termes du sujet filtré, alors qu'ici, le nombre d'étapes dépend en plus de la structure des motifs, ce qui le rend bien souvent inférieur au nombre de sous-termes du sujet. Les résultats obtenus en pratique montrent que le raffinement glouton permet de réduire le nombre de tentatives de filtrage dans une proportion variant entre 30% et 85%.

6.7 Calcul des substitutions

Une fois la première étape de filtrage effectuée, il reste à calculer les instances des variables du membre gauche de la règle pour pouvoir évaluer les conditions et construire le terme résultat. Deux problèmes méritent d'être considérés : comment instancier les variables non introduites dans les graphes bipartis ? comment optimiser la construction des substitutions associées aux autres variables ?

Instanciation des variables apparaissant sous un symbole AC

Les variables qui apparaissent directement sous un symbole AC du motif ne sont pas prises en compte par les étapes précédentes. Lorsqu'il y a seulement une ou deux variables (avec des multiplicités), il n'est pas nécessaire de construire un système d'équations diophantiennes pour calculer leur instance. Différents cas peuvent être étudiés en fonction de la structure syntaxique du motif.

- pour un motif de la forme $f_{AC}(x_1,t_1,\ldots,t_n)$, une fois que les sous-termes du sujet ont été filtrés par les $t_1,\ldots,t_n$, tous les sous-termes du sujet, non « capturés » par un t_i (*i.e.* n'intervenant pas dans une solution du graphe biparti) sont associés à x_1 ;
- pour $f_{AC}(x_1,x_2^{\alpha_2},t_1,\ldots,t_n)$, considérons dans un premier temps le cas où $\alpha_2 = 1$. Une fois que les sous-termes du sujet ont été filtrés par les $t_1,\ldots,t_n$, les sous-termes non capturés sont partitionnés en deux ensembles de toutes les façons possibles. Un ensemble est utilisé pour instancier x_1 et l'autre pour x_2.
  Lorsque $\alpha_2 > 1$, après l'étape de filtrage, on cherche toutes les façons d'associer α_2 sous-termes identiques non capturés. Les sous-termes restant étant associés à x_1 ;
- pour $f_{AC}(x_1,x_2^{\alpha_2},g_{AC}(x_3,x_4^{\alpha_4}))$, un sous-terme $\hat{s}_i$ du sujet est filtré par g_{AC}. Les sous-termes de s_i sont divisés en deux ensembles (tel que décrit précédemment) qui sont associés aux variables x_3 et x_4. Les sous-termes du sujet non capturés ($\{s_j \mid j \neq i\}$) sont eux aussi partitionnés et associés aux variables x_1 et x_2, comme décrit précédemment.

Compilation de la construction des substitutions

Rappelons que notre objectif est de réaliser un compilateur et que nous essayons, dans la mesure du possible, de traduire les structures de données de l'algorithme en des structures de contrôle du langage cible pour réduire au minimum le nombre d'allocations mémoire effectuées au cours de l'exécution du programme généré.

Dans le cas syntaxique, le fait d'avoir au plus une substitution à construire rend facile leur construction : il suffit d'utiliser l'automate de filtrage et d'associer à chaque état de l'automate une *variable* (du langage cible) permettant de mémoriser les termes lus lorsqu'une règle de transition d'états $\delta_i : (e,\omega) \longrightarrow_\delta e'$ (une arête étiquetée par un ω) est utilisée. Dans le cas AC, il peut y avoir plusieurs instanciations différentes pour une même variable du problème de filtrage considéré. Il ne devient donc plus possible de réserver un nombre fixe d'emplacements pour mémoriser les instances des variables « traversées » par l'automate de filtrage. Il faudrait créer dynamiquement une structure de données capable de mémoriser toutes les instances possibles, mais cela deviendrait trop coûteux. De plus, la construction d'une telle structure dynamique n'est pas nécessaire lorsque la première règle sélectionnée peut s'appliquer : toutes les substitutions mémorisées sont alors détruites.

Notre approche consiste à construire la substitution seulement après avoir résolu le problème de filtrage. Pour chaque sous-motif $p_{j,k}$, les positions des variables sont connues et utilisées

lors de la compilation. Nous pouvons utiliser cette information pour construire une fonction d'accès $access_p_{j,k}$ qui prend un terme clos en argument et retourne la liste des instances des variable de $\hat{p}_{j,k}$ (notons que la taille de cette liste est fixée et dépend seulement du $p_{j,k}$ considéré). Étant donnée une solution $S_j = \{[\hat{s}_i,\hat{p}_{j,k}]\}$ du graphe BG_j, l'ensemble des instances $I_j = \{access_p_{j,k}(s_i) \mid [\hat{s}_i,\hat{p}_{j,k}] \in S_j\}$ peut être calculé.

Considérons les règles $f_{AC}(f(a,x),f(y,g(b))) \rightarrow r_2$ et les fonctions $access_f(a,x)(t) = t_{|2}$ et $access_f(y,\ g(b))(t) = t_{|1}$. Partant de la solution $S_2 = \{[f(a,a),f(a,x)],\ [f(a,g(b)),f(y,g(b))]\}$, l'ensemble $I_2 = \{a,a\}$ est facilement calculé pour construire les instances de x et y : $\sigma = \{x \mapsto a, y \mapsto a\}$. Une implantation de ces fonctions d'accès est donnée par le programme 6.3.

Compilation des fonctions d'accès

```
void variable_extract_F(struct term *subject, int id_pattern,
                        struct term *substitution[], int indice) {
  switch(id_pattern) {
  case 0: /* f(a,x) */
    substitution[indice]=subject->subterm[1];  (indice)++;
    break;
  case 1: /* g(a) */
    break;
  case 2: /* f(y,g(b)) */
    substitution[indice]=subject->subterm[0];  (indice)++;
    break;
  }
}
```

Programme 6.3: *Étant donné un numéro de motif* **`id_pattern`** *(0, 1 ou 2 dans cet exemple) et un terme clos* **`subject`***, cette fonction récupère et mémorise (dans le tableau* **`substitution`***) les instances des variables qui apparaissent dans le motif correspondant.*

6.8 Extension à l'ensemble des motifs

Bien qu'utile en pratique, l'approche présentée jusqu'ici n'a d'intérêt que pour une certaine classe de motifs décrite au paragraphe 6.4. Deux possibilités doivent alors être envisagées afin de pouvoir traiter les règles dont le membre gauche n'est pas dans C_2. La première consiste à étendre notre algorithme de filtrage pour le rapprocher de celui décrit au paragraphe 6.3, au risque de voir la complexité du processus de compilation s'accroître considérablement : établir une coopération entre des structures dynamiques complexes (hiérarchies de graphes bipartis par exemple) et des structures de contrôle elles aussi complexes (les hiérarchies d'automates de filtrage par exemple), n'est pas forcément aisé. Dans notre projet, il est primordial que le code généré par le compilateur soit correct et que le fonctionnement du compilateur soit lui-même relativement simple pour permettre son extension et un développement en équipe. L'intégration d'une procédure complexe est certes intéressante mais pas forcément un bon choix si elle ne permet d'améliorer que des situations se présentant rarement et si elle risque de compromettre le développement de l'outil.

C'est pourquoi nous avons retenu une deuxième approche qui consiste à transformer les

règles dont le membre gauche n'appartient pas à C_2 en des règles équivalentes ayant un membre gauche compilable par notre algorithme. N'importe quelle règle $l \to r$ (avec $l \notin C_2$) peut être transformée en une règle $l' \to r$ utilisant des *conditions de filtrages* (présentées au paragraphe 1.4) et telle que l' appartienne à C_2. La transformation présentée ci-dessous se décompose en deux cas, suivant que le symbole de tête de l est AC ou non.

- Soit $l = f_{AC}(x_1^{\alpha_1}, \ldots, x_m^{\alpha_m}, t_1, \ldots, t_k, t_{k+1}, \ldots, t_n)$ avec $x_1, \ldots, x_m \in \mathcal{X}$, $\alpha_j \geq 0$, $t_1, \ldots, t_k \in C_1$ et $t_{k+1}, \ldots, t_n \notin C_1$, où $k < n$.
  Si $l' = f_{AC}(x_1^{\alpha_1}, y, t_1, \ldots, t_k)$. La règle

  $$l' \to r \textbf{ where } f_{AC}(x_2^{\alpha_2}, \ldots, x_m^{\alpha_m}, t_{k+1}, \ldots, t_n) := y$$

  est équivalente à la règle précédente. Rappelons ici qu'au cours de l'évaluation d'une condition de filtrage, la variable y est instanciée par la substitution qui permet à l' de filtrer le sujet.
- Soit $l = f(t_1, \ldots, t_n)$ avec des $t_i \notin C_2$. Soit Λ une fonction d'abstraction qui remplace des sous-termes u_j de l par une nouvelle variable x_j, $j = 1, \ldots, k$, de sorte que $l' = \Lambda(l) \in C_2$. (Pour se convaincre de l'existence d'une telle fonction Λ : il suffit de considérer tous les sous-termes u_j dont la racine est un symbole AC. Dans ce cas extrême, on a même $l' = \Lambda(l) \in C_0$.)
  Considérons la nouvelle règle

  $$\begin{aligned} l' \to r \quad & \textbf{where } u_1 := x_1 \\ & \vdots \\ & \textbf{where } u_k := x_k \end{aligned}$$

  qui est bien équivalente à $l \to r$.

Deux règles sont équivalentes lorsque les membres droits sont identiques et que l'ensemble des substitutions permettant d'appliquer les règles sont identiques. On dit qu'une substitution permet d'appliquer une règle si le membre gauche de la règle, instancié par la substitution, filtre bien le sujet et si l'ensemble des évaluations locales (conditions et conditions de filtrage) sont satisfaites lorsqu'elles sont instanciées par la substitution.

Considérons deux opérateurs AC $\cup_{AC}$ et eq_{AC}, un constructeur e et un terme $r(x_1,x_2,x_3)$ utilisant les trois variables x_1,x_2,x_3. La règle :

$$x_1 \cup_{AC} eq_{AC}(\mathsf{e}(x_2),\mathsf{e}(x_3)) \to r(x_1,x_2,x_3)$$

ne peut pas être directement traitée par notre algorithme parce que le sous-terme $eq_{AC}(\mathsf{e}(x_2),\mathsf{e}(x_3))$ n'appartient pas à la classe C_0. Cependant, en introduisant une nouvelle variable y et une condition de filtrage **where** $eq_{AC}(\mathsf{e}(x_2),\mathsf{e}(x_3)) := y$, la règle suivante devient équivalente à la précédente et son membre gauche appartient à C_1:

$$x_1 \cup_{AC} y \to r(x_1,x_2,x_3) \quad \textbf{where } eq_{AC}(\mathsf{e}(x_2),\mathsf{e}(x_3)) := y$$

La transformation appliquée correspond au premier schéma proposé, mais on aurait pu appliquer le deuxième schéma et obtenir un membre gauche appartenant à la classe C_2 :

$$\begin{aligned} x_1 \cup_{AC} eq_{AC}(y_2,y_3) \to r(x_1,x_2,x_3) \quad & \textbf{where } \mathsf{e}(x_2) := y2 \\ & \textbf{where } \mathsf{e}(x_3) := y3 \end{aligned}$$

L'intérêt de cette dernière transformation étant d'introduire des problèmes de filtrage syntaxiques dans les condtions de filtrage et de profiter au maximum des algorithmes de compilation du filtrage AC présentés dans ce chapitre.

Considérons maintenant la règle de réécriture suivante :

$$\mathsf{solve}(\mathsf{simplify}(x_1 \cup_{AC} eq_{AC}(\mathsf{e}(x_2),\mathsf{e}(x_3)))) \rightarrow r(x_1,x_2,x_3)$$

Le membre gauche commence par un symbole syntaxique et le sous-terme $\mathsf{simplify}(x_1 \cup_{AC} eq_{AC}(\mathsf{e}(x_2),\mathsf{e}(x_3)))$ n'est pas dans C_2, ce qui nous amène à appliquer le deuxième schéma de transformation proposé : considérons la fonction d'abstraction $\Lambda = \{\mathsf{e}(x_2) \mapsto y_2,\mathsf{e}(x_3) \mapsto y_3\}$. Le membre gauche de la règle suivante appartient désormais à la classe C_2 :

$$\begin{array}{ll} \mathsf{solve}(\mathsf{simplify}(x_1 \cup_{AC} eq_{AC}(y_2,y_3)) \rightarrow r(x_1,x_2,x_3) & \textbf{where } \mathsf{e}(x_2) := y2 \\ & \textbf{where } \mathsf{e}(x_3) := y3 \end{array}$$

Il est intéressant de noter qu'au cours de l'évaluation d'une condition de filtrage, seul un algorithme *one-to-one* est nécessaire. Lorsqu'un motif u_j contient des symboles AC, nous utilisons une procédure de filtrage générale telle que celle présentée au paragraphe 6.2 et décrite en détail dans (Eker 1995). Cela signifie que dans le pire des cas, notre algorithme de filtrage AC many-to-one sert à effectuer une pré-sélection fondée sur la couche syntaxique supérieure des membres gauches de règles et que les problèmes de filtrage AC sont résolus par un algorithme one-to-one. C'est précisément l'approche suivie dans l'implantation de Maude (Clavel, Eker, Lincoln et Meseguer 1996).

6.9 Synthèse

Les travaux présentés dans ce chapitre sont certes théoriques mais leurs apports sont principalement pratiques. Le cœur de la méthode de compilation proposée repose sur la définition d'une classe restreinte de termes comprenant les motifs qui apparaissent le plus souvent en pratique. La limitation imposée sur le nombre de symboles AC imbriquées évite la résolution d'équations diophantienne et permet d'utiliser une structure compacte de graphes bipartis qui accélère le traitement des règles conditonnelles. Afin d'aboutir à une procédure de nomalisation AC efficace, nous prenons en compte, dès la conception de l'algorithme de filtrage, les problèmes liés à l'extraction des solutions et à la construction des substitutions. Nous proposons ainsi des techniques de compilation du filtrage et de la normalisation AC dont les résultats expérimentaux sont présentés au chapitre 12. Enfin, le cas général est traité par une transformation de programmes permettant de se ramener à la classe de motifs définie.

Chapitre 7

Gestion du non-déterminisme

7.1 Introduction 100
7.2 Basic choice point primitives 101
7.3 Known choice point implementations 102
7.4 New choice point management 103
7.5 Imperative programming with backtracking 109
7.6 Concluding Remarks 111

Les caractéristiques d'ELAN sont telles que des aspects non-déterministes apparaissent à toutes les étapes composant l'application d'une règle (sélection d'une règle au cours du filtrage syntaxique, résolution des graphes bipartis et instanciation des variables au cours du filtrage AC, évaluation des conditions de filtrage et application des stratégies au cours du calcul des évaluations locales). Il devient alors essentiel de définir un mécanisme uniforme de gestion du non-déterminisme, qui puisse être utilisé de manière cohérente par toutes ces étapes.

Pour des raisons d'efficacité, nous avons volontairement choisi d'utiliser le langage C comme langage cible du compilateur ELAN. Ce choix rend relativement difficile la gestion du non-déterminisme, simplement parce qu'aucun mécanisme adéquat n'est prévu dans ce langage. Il existe cependant différentes façons de gérer le non-déterminisme et plus particulièrement la pose de points de choix en C, mais les approches connues dénaturent souvent l'utilisation « conventionnelle » du langage C en interdisant l'utilisation d'arguments lors des appels de fonctions, ce qui oblige le programmeur à gérer explicitement une pile d'arguments et de variables locales.

Comme le montrent les chapitres 8 et 11, des schémas complexes de compilation des stratégies sont étudiés et certaines fonctions nécessaires au filtrage AC sont amenées à être écrites par un humain, ce qui donne une importance supplémentaire à la lisibilité du code généré et à la facilité d'utilisation des fonctions de gestion du non-déterminisme.

C'est pourquoi nous avons choisi de ne pas « dénaturer » l'utilisation du langage C et de lui ajouter deux nouvelles primitives, setChoicePoint et fail, qui n'imposent aucune restriction : la compilation modulaire, l'utilisation de bibliothèques extérieures, l'usage de variables locales et de fonctions avec arguments restent possibles.

Ces deux fonctions permettent respectivement de poser un point de choix qui mémorise l'état courant de l'exécution (la valeur des variables locales) et d'y revenir ultérieurement en restaurant l'état sauvegardé. Leur action se situe au niveau de la pile système gérée par les compilateurs C, ce qui nous oblige à intervenir à bas niveau et en particulier à écrire ces deux fonctions en assembleur.

Ce chapitre présente dans un premier temps le comportement de ces deux fonctions et montre comment elles peuvent être utilisées pour programmer facilement des retours arrière en C. Dans une deuxième partie, les algorithmes sont décrits en détail pour permettre une éventuelle amélioration ou modification de l'approche proposée. La lecture attentive de ce chapitre n'est pas essentielle à la bonne compréhension des schémas de compilation d'ELAN en général, et peut être évitée en première lecture. Le lecteur doit cependant s'assurer qu'il comprend bien l'effet de ces deux primitives sur l'exécution d'un programme, sans pour autant savoir comment elles s'implantent.

Le contenu de ce chapitre est particulier dans la mesure où il est technique et peut être lu indépendamment de tous les autres. C'est pourquoi il nous a semblé inutile de le traduire en français. Le texte ci-dessous a été intégralement publié dans (Moreau 1998a).

A choice-point library for backtrack programming

Abstract

Implementing a compiler for a language with nondeterministic features is known to be a difficult task. This paper presents two new functions setChoicePoint and fail that extend the C language to efficiently handle choice point management. Originally, these two functions were designed to compile the ELAN strategy language. However, they can be used by programmers for general programming in C. We illustrate their use by presenting the classical 8-queens problem and giving some experimental results. Algorithms and implementation techniques are sufficiently detailed to be easily modified and re-implemented.

7.1 Introduction

In the area of formal specifications, rewriting techniques have been developed for two main applications: prototyping algebraic specifications of user-defined data types and theorem proving related to program verification. In this context we are interested in nondeterministic computation and deduction.

Term rewriting is nondeterministic in the sense that there may be several reductions starting from one initial term and producing different results. Rewriting logic (Meseguer 1992) gives a logical background and raises new interesting problems concerning the efficient implementation of nondeterministic rewriting which needs backtracking. This is similar to the implementation of logic programming languages, but a significant difference is the fact that rewriting rules can be applied inside the terms. Moreover the formalism used to prune the search space is different from that of logic programming languages.

In this paper we present a new technique for compiling the specific control flow in programs during the backtracking. Our method preserves the efficiency of deterministic computations and is of more general interest; it could be used in implementations of constraint solvers, imperative languages with backtracking such as Alma-0 (Partington 1997, Apt et Schaerf 1997), the WAM (Warren 1983, Aït-Kaci 1990) and Prolog-like languages.

A first implementation of our techniques has been done by Marian Vittek in 1996. The experimental results, presented in (Vittek 1996), show that nondeterministic rewriting can be implemented as efficiently as the best current implementations of functional and logic programming languages. This paper presents a formalisation of the implementation and gives detailed algorithms to re-use, adapt and improve the proposed method. It took great benefit from the idea and comments of Marian Vittek.

Section 7.2 illustrates the behaviour of usual functions used to implement backtracking in nondeterministic computations. Section 7.3 gives a brief overview of existing techniques for implementing choice points and compiling languages with nondeterministic features into C. Section 7.4 presents algorithms for the two new proposed functions that implement an efficient backtracking control flow: setChoicePoint and fail. Then Section 7.5 illustrates on one example how the use of setChoicePoint and fail can help in solving in a natural way algorithms that involve search. Some experimental results show that the proposed method can be a good alternative to compile in an efficient way languages that involve nondeterministic features.

7.2 Basic choice point primitives

Backtracking is a well-known approach to implement nondeterministic computations. In compilation techniques, two functions are usually needed: the first one to create a choice point and save the execution environment. The second one to backtrack to the last created choice point and restore the saved environment. Many languages that offer nondeterministic capabilities provide similar functions: for instance world+ and world- in Claire (Caseau et Laburthe 1996), try and retry in the WAM, onfail, fail, createlog and replaylog in the Alma-0 Abstract Machine (Partington 1997), setChoicePoint and fail in ELAN (Vittek 1996). Recently, a new approach to the implementation of tabling for Prolog (Demoen et Sagonas 1998) has been proposed. The authors suggest to extend a Prolog implementation by adding some new built-in predicates. For their purpose, a similar idea as the one presented here has been explored in a different context.

We propose to extend the C language by adding two control flow functions: setChoicePoint and fail. setChoicePoint returns the integer 0 when setting a choice point, and the computation goes on. When the function fail is called, it performs a jump into the last call of setChoicePoint and it returns the integer 1. These functions can remind the pair of standard C functions `setjmp` and `longjmp`. However, the `longjmp` can be used only in a function called from the function setting `setjmp`. Functions setChoicePoint and fail do not have such a limitation.

The following program, written in C, illustrates the behaviour of these two new functions. The **Output** column shows the result obtained when executing the program:

Program	**Output**

`static int counter=0;`
`main() {`
`  if(setChoicePoint()!=0) exit(0);`
`  f();`
`  fail();`
`}`
`f() {`
`  int result, locvar=0;`
`  result=setChoicePoint();`
`  printf(result,locvar,counter);`
`  locvar++; counter++;`
`  printf(locvar,counter);`
`}`
	`result=0, locvar=0, counter=0`
	`locvar=1, counter=1`
	`result=1, locvar=0, counter=1`
	`locvar=1, counter=2`

When setting a choice point, only local variables are saved. If a failure occurs, only local variables are restored to the value they had when setting the choice point. When executing the

example program, a first choice point is created and the computation goes on. The function `f()` is called and `locvar` is initialised to 0. Then a second choice point is created, `result` and `locvar` are saved, printed, incremented and printed again. Before executing the first fail, `counter=1` and `locvar=1`. Then a backtrack is performed: the function fail restores the last saved environment (`f` is re-activated, `locvar=0`) and transfers the control to setChoicePoint function which returns the integer 1. This explains why the third line is `result=1`, `locvar=0`, `counter=1`. The function fail is called again: a backtrack to the first set choice point is performed; the conditional test is evaluated to true and the program stops.

7.3 Known choice point implementations

The implementation of choice point management most often involves two mechanisms: first, an environment stack, called the *trail*, to save local variable values and a continuation address; second, a control flow handler to perform the jump to the saved continuation address when backtracking.

A number of techniques for implementing branching schemes have been proposed over the years, especially in the functional and logic programming communities. Several languages use C as target language such as Cg, Icon, Janus, Erlang, KL1, RML and Mercury. Their different compilation schemes are presented in (Budd 1982, Wampler et Griswold 1983, Demoen et Maris 1994, Codognet et Diaz 1995, Pettersson 1995, Henderson, Conway et Somogyi 1996).

Among these techniques, the simplest method implements branching using a C `goto` statement. However problems arise because indirect branching is not available in standard C and also because a `goto` instruction can only do a jump into its function scope. This leads to a C program composed of a unique huge function with a `switch` statement to simulate indirect gotos. This compilation scheme is unrealistic since it makes impossible separate compilation. Moreover, collecting all codes into one C function affects compilation time and compiler's ability to perform register allocation.

The second method consists in translating each labelled block by a C function that returns a continuation address. Those functions are managed by a driver function that does the necessary dispatching to transfer control from one function to another. Consequently, this method is not the most optimised one but is suited for standard C and separate compilation.

A third well-known existing scheme consists in using non standard C features that are supported by the GNU C compiler (Stallman 1995). The `gcc` compiler makes it possible to take the address of labels, and later on to jump to those addresses. It also offers the possibility to insert inline assembly code, and to specify the assembly name of a function. With those extensions it is now possible to translate any branching by a `goto` statement.

When using one of the three presented schemes, the structure of a program is not taken into account. Parameter passing cannot be done in a natural way. Instead, global variables are used to communicate arguments from caller to callee. That is why local variables have to be saved before doing a jump and classical function calls have to be simulated. As a matter of fact, it is very difficult for a human to write a program in these conditions and the presented compilation schemes can only be used in automatically generated programs.

Even if the three presented methods are efficient and well-designed to implement a WAM-like abstract machine, it is still difficult to use them to design new compilation schemes because resulting programs are often difficult to read.

7.4 New choice point management

In this section we present algorithms and implementation techniques of the two new functions setChoicePoint and fail.

The key idea of our approach is to use the system stack and only one environment stack to store values that have to be saved. Consequently, there is no restriction on the usage of local variables and parameter passing when programming with setChoicePoint and fail.

We first present an approach which consists in extending the two standard C functions `setjmp` and `longjmp`. Then we define some notations in order to give detailed algorithms of the second method which minimises the size of memory blocks that have to be saved (resp. restored) when setting a choice point (resp. performing a failure).

7.4.1 setJump: an extension of setjmp

The standard C library defines two low level functions `setjmp` and `longjmp`. The first one saves the current execution context (machine registers and a return address) in a `jmp_buf` structure. The second one can restore any stack context that has been saved in a `jmp_buf` structure by `setjmp`. After the `longjmp` runs, the program execution continues as if the corresponding call to the `setjmp` function had just returned the value specified in the `longjmp` call. The result of `longjmp` is undefined if the function that made the corresponding call to the `setjmp` has already returned.

We propose first to extend `setjmp` and `longjmp` into setJump and longJump to suppress such undefinedness: the whole stack system (memory block between the *base pointer* and the *stack pointer*) has to be saved in a Jump_buf structure when calling setJump.

Given an integer different from 0 and a valid Jump_buf structure, the longJump function restores registers and the whole system stack, and then the integer parameter is returned.

Depending on the architecture, these two functions may be implemented in C: `setjmp` and `longjmp` are used to save and restore registers[2] and `memcpy` is used to copy memory blocks. This approach was successful on a PC under Linux and a DEC Alpha-Station but the result is not really safe: as mentioned previously, `longjmp` is used in a non-standard way; since it is not possible in C to get the *base pointer* and the *stack pointer*, some heuristics are used; and furthermore, the behaviour is not stable when using advanced optimisation options such as `gcc -O6`. This is why we recommend this approach only to get a first easy implementation, but to get a safe implementation, setJump and longJump have to be re-implemented in assembly language. There is nothing surprising, because `setjmp` are `longjmp` are themselves implemented in assembly language.

Some processors, such as Sparc, are based on a window register architecture. In this case, it is not possible to save and restore the window position: the corresponding assembly instruction must be executed in privileged mode (not accessible by users). This restriction makes impossible the implementation of setJump and longJump on such architecture, however, in Section 7.4.4, we present a general algorithm for setChoicePoint and fail that can be implemented on almost any architectures.

7.4.2 A first implementation of setChoicePoint and fail

setChoicePoint and fail are somehow restrictions of setJump and longJump because the fail function always restores the context of the last set choice point. This special case allows us to design

[2] A similar idea is used in the BDW Garbage Collector (Boehm et Weiser 1988).

smarter and more efficient algorithms. A naive implementation of setChoicePoint and fail consists in reusing setJump, longJump implementations and storing Jump_buf structures in a LIFO data structure (the *trail* itself).

Let us remark that saving the whole system stack is too expensive. In average, only a small part of the system stack needs to be changed when a failure occurs. For example, let us consider the following program, where dots denote irrelevant instructions:

```
void main() {
  ...
  g(i);
  ...
  fail();
  ...
}
```

```
void g(int arg) {
  ...
  setChoicePoint();
  ...
}
```

An execution of the `main` function creates the stack frame of `main` in the system stack. Then `main` calls `g`, this pushes the stack frame of `g` onto the stack. So, when the choice point is set, two stack frames (`main` and `g`) are on the stack (see Figure 7.1). After this, when leaving `g`, its stack frame is freed and execution continues in `main` by fail. But, at this moment, the stack contains the stack frame of `main`. So, only the stack frame of `g` has to be restored onto the current system stack to reconstitute the stack as it was at the moment of the choice.

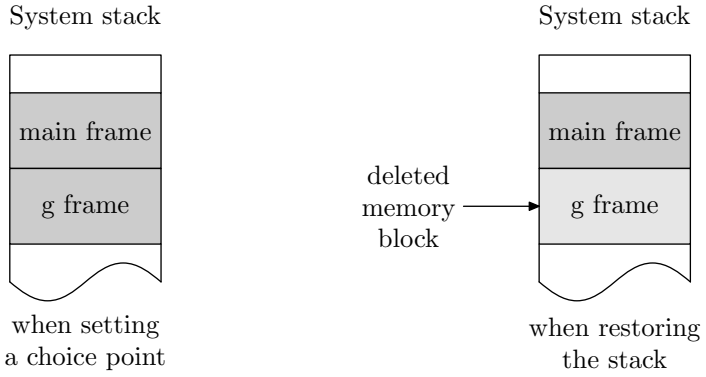


Figure 7.1: It is useless to copy the whole system stack

This example illustrates the fact that this is useless to copy the whole system stack because only the stack frames of some functions are concerned. The next implementation of setChoicePoint uses this idea.

7.4.3 Notations

In order to give a detailed algorithm several notations are needed. They are useful to clearly compute memory blocks that have to be saved and restored.

Let us first consider a simple execution model that consists in viewing a program execution as a sequence of instruction executions, function calls, and function returns.

Let us define watch points $\tau_1, \ldots, \tau_{n_\tau}$ as the first executed instruction after a function call or a function return.

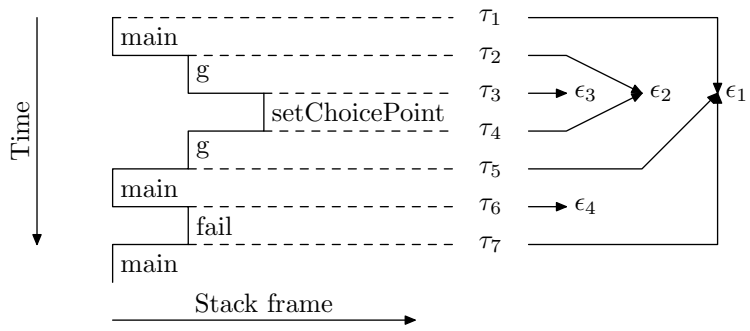


Figure 7.2: Setting watch points and environments

When running the program presented in Section 7.4.2, the function `main` calls `g` which calls setChoicePoint. The first executed instruction when entering `main`, `g` and setChoicePoint corresponds to watch points τ_1, τ_2 and τ_3 respectively. The first executed instruction after setChoicePoint corresponds to τ_4. Other watch points τ_5, τ_6 and τ_7 are defined similarly. The program execution and watch-points are illustrated in Figure 7.2.

When executing a function, there is a corresponding environment called ϵ_j that contains information about the current executing function. This information is available through different functions:

- $fp(\epsilon_j)$ to get the frame pointer value, i.e. the address which indicates the beginning of the stack frame;
- $ra(\epsilon_j)$ to get the return address value, i.e. the address from the program to which the program counter should be restored;
- local variables are also saved in an environment.

To each watch point τ_i is associated an environment ϵ_j (i and j are not equal in general because several watch-points may be associated to a given environment). The notion of environment and the correspondences between (τ_i) and (ϵ_j) are illustrated in Figure 7.2.

A stack pointer value $sp(\tau_i)$, which is the current top of the system stack, is associated to each watch point τ_i.

Let us define Env_at as the surjective function from $\{\tau_1, \ldots, \tau_m\}$ to $\{\epsilon_1, \ldots, \epsilon_n\}$ that maps each (τ_i) to its environment (ϵ_j). The set $\{\tau_1, \ldots, \tau_m\}$ is totally ordered by indices values: $\tau_1 < \cdots < \tau_m$.

In the previous example we have:

- $Env_at(\tau_1) = Env_at(\tau_5) = Env_at(\tau_7) = \epsilon_1$;
- $Env_at(\tau_2) = Env_at(\tau_4) = \epsilon_2$;
- $Env_at(\tau_3) = \epsilon_3$;
- $Env_at(\tau_6) = \epsilon_4$.

Environments are organised as in a block structure language, thus, the notion of *embedded* environment can be defined: for a given ϵ_j, the embedded environment $emb(\epsilon_j)$ is the environment in which the function associated to ϵ_j is called.

In the previous example, we have:

- $emb(\epsilon_3) = \epsilon_2$
- $emb(\epsilon_2) = emb(\epsilon_4) = \epsilon_1$
- $emb(\epsilon_1)$ is not defined

Let us define for $\epsilon \in \{\epsilon_1, \ldots, \epsilon_n\}$ the function that returns the minimum element of the inverse image of Env_at: $Min(\epsilon) = min(Env_at^{-1}(\epsilon))$

From a practical point of view, τ_i is a program's address that contains an assembly instruction. ϵ_j is a stack frame associated to the current executed C function and $Min(\epsilon)$ is the first executed instruction when entering a C function.

Let $<$ be a total ordering on addresses such that the base pointer bp is the minimum. Let l_1, l_2 be two addresses of the system stack. If $l_1 < l_2$, l_1 is said to be closer to bp than l_2. When $l_1 < l_2$, $[l_1, l_2[$ is the memory block between those two addresses.

7.4.4 Advanced algorithm for setChoicePoint and fail

The goal of the algorithm sketched in Section 7.4.2, is to minimise the number of saved stack frames. The main idea consists in implementing a special *handle function* which saves the top stack frame of the system stack.

Each time a nondeterministic function[3] returns to the caller function, the last jump is redirected to this *handle function*: return addresses of nondeterministic functions are successively (first by setChoicePoint and then by the handle function itself) modified to point to this handle function. This guarantees that the handle function is called each time a nondeterministic function executes the return instruction. These calls save the corresponding stack frames which are then used to recover the original system stack by the fail function. The setChoicePoint algorithm performs two actions: save machine registers and activate the handle function.

Let τ_i be a choice point: setChoicePoint saves registers, which include $ra(Env_at(\tau_i))$, $sp(\tau_i)$ and $fp(Env_at(\tau_i))$, pushes the special mark endReg into the trail and then jumps into the handle function: saveFrame.

Let ϵ_j be the environment of the function which did the branching to saveFrame (when creating the choice point, $Env_at(\tau_i) = \epsilon_j$):

- saveFrame saves the frame of the function that called the function associated to ϵ_j. Let us call $emb(\epsilon_j)$ its environment. The saved frame is $[fp(emb(\epsilon_j)), sp(Min(\epsilon_j))[$. Then saveFrame pushes the special mark endFrame into the trail,
- saveFrame replaces the caller's return address (saved in $emb(\epsilon_j)$) by the saveFrame procedure's address. Thus, each time a function returns, the save frame handler is activated.

The handle function saveFrame may be called several times before a fail occurs. In this case, several frames are saved into the trail. Figure 7.3 illustrates this possibility: two choice points have been created and three frames have been saved (two of them are associated to the first choice point). This situation occurs when the function calling setChoicePoint had returned.

The fail algorithm rebuilds the system stack with saved frames until the endReg code is found. Then, the registers and the stack pointer are restored, and the integer 1 is returned.

[3] A function is said to be nondeterministic if a choice point is set or a nondeterministic function is called during its execution.

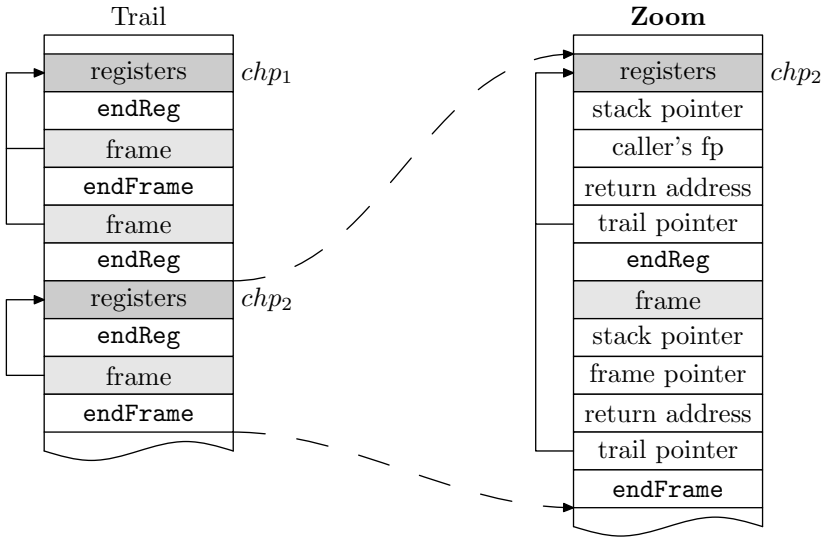


Figure 7.3: Trail stack state after setting a choice point

Note that fail removes the last created ChoicePoint and restores the system stack to its initial state. Assuming that a return address is saved in the system stack, the return address (that was modified to saveFrame) is restored to its initial value by recovering the stack. Consequently, the save frame handler is no longer active.

7.4.5 Detailed implementation

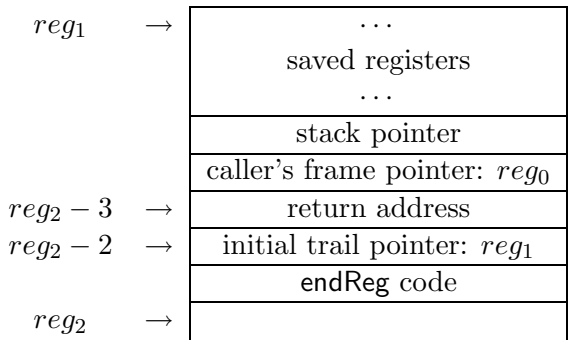
In this section we give a low level description that corresponds to the assembly language implementation. The next algorithm describes the implementation of setChoicePoint:

Algorithm 7.1 setChoicePoint

load the caller's frame pointer ($fp(Env_at(\tau_i))$) into reg_0
load the trail pointer into reg_1 and reg_2
push non specific registers into the trail (referenced by reg_2)
push the stack pointer (sp), the caller's frame pointer reg_0, the return address (ra) and the initial trail pointer reg_1 into the trail
push the endReg code
prepare the return value: 0
do a jump to saveFrame

The following figure illustrates the state of the trail stack after executing setChoicePoint.

This corresponds to the top level of the zoom part given in Figure 7.3.



This saved information is used by the handle function saveFrame to save stack frames and update links to the corresponding choice point. Note that the notation $reg_2 - 3$ is used to specify a base address. In this example, the value 3 is subtracted from the base register reg_2 value to denote a new address whose contents is the saved return address.

The following algorithm describes a pseudo assembly code of the saveFrame handle function:

Algorithm 7.2 saveFrame

load the trail pointer into reg_2
load the caller's frame pointer into reg_0
restore the saved return address $(reg_2 - 3)$ into reg_4
restore the saved trail pointer $(reg_2 - 2)$ into reg_5
push the frame (memory block $[reg_0, sp[$) into the trail
push the stack pointer and caller's frame pointer reg_0
load the frame pointer of the previous memory block $(reg_5 - 4)$ into reg_3
if $reg_3 = reg_0$ **then**
  {the return address is already into the trail. The link to the previous block has to be followed}
  load the old return address $(reg_5 - 3)$ into reg_4
  load the beginning of the previous block $(reg_5 - 2)$ into reg_5
end if
push reg_5 and reg_4 and the endFrame code into the trail
update the trail pointer
modify the caller's return address to the save frame handler saveFrame

The state of the trail stack after executing saveFrame is described in Figure 7.3. Let us notice that top and bottom parts of the zoom stack have similar structures: values are saved in the same order.

The following pseudo assembly code describes the implementation of fail:

Algorithm 7.3 fail

load the trail pointer into reg_2
`recoverFrame:` load the current code $(reg_2 - 1)$
if it is a endReg code **then**
  branch to `returnInCP`
end if
restore the stack pointer $(reg_2 - 5)$
restore the frame pointer $(reg_2 - 4)$
compute the saved frame size and restore it
branch to `recoverFrame`
`returnInCP:` update the trail pointer
restore saved registers, including the return address register
return the value 1

7.4.6 Portability

The two functions setChoicePoint and fail are implemented with a 200 lines assembly library. This could be a source of difficulty for portability. However, in practice, the library can be easily implemented on a new architecture. In fact, only three specific access functions are required: $fp()$, $sp()$ and $ra()$ which return respectively the current frame pointer, the current stack pointer and the return address. The nondeterministic library[4] has been implemented on three different architectures: Sparc, Intel and DEC-Alpha. It has not been ported yet to HP-PA or MIPS processors, but it should not be a problem even if the stack grows from low to high addresses.

It is well-known that efficient implementations of nondeterministic library usually take benefit of `gcc` extensions and use inline statements to add assembly labels in the generated code. Therefore, our approach is not less portable than classical implementations of choice point managements.

7.5 Imperative programming with backtracking

Several languages such as 2LP (McAloon et Tretkoff 1995) and Alma-0 (Partington 1997, Apt et Schaerf 1997) have been developed to combine advantages of logic and imperative programming in order to deal in a natural way with algorithmic problems that involve search. Alma-0 extends imperative programming with some features inspired by the logic programming paradigm. For instance:

- use of boolean expressions as statements and vice versa;
- the `ORELSE` statement starts by proceeding through the first branch. If the computation eventually fails, backtracking takes place and the computation resumes with the next branch in the state in which the previous branch was entered;
- the `SOME` statement extends the `ORELSE` construction to generate a number of choice points determined only at run-time;
- the `FORALL` statement introduces a controlled form of iteration over the backtracking.

[4] Available at `http://www.loria.fr/ELAN`.

With these extensions, the program that computes all solutions of the N-queens problem can be expressed in a natural way. Let us remind that the N-queens problem consists in placing N queens on the chess board so that they do not attack each other.

```
FOR column := 1 TO N DO
  SOME row := 1 TO N DO
    FOR i := 1 TO column-1 DO
      x[i] <> row;
      x[i] <> row+column-i;
      x[i] <> row+i-column
    END;
    x[column] = row
  END
END
```

```
BEGIN
(* print all solutions *)
  nrSols := 0;
  FORALL queens(b);
    DO
      (* print solution: b *)
    END;
END queens.
```

In the Alma-0 environment, the program is first translated into the Alma-0 abstract machine language (AAA), and then, each AAA instruction is compiled into C statements. The used backtracking mechanism corresponds to the first method described in Section 7.3. Thus, modular compilation is not possible and optimisations performed by the C compiler are not optimal.

We think that setChoicePoint and fail could be a good alternative to implement AAA backtracking operations. Even more, setChoicePoint and fail seem to be well-suited to compile directly an Alma-0 program to C without any intermediate abstract machine. Indeed, the previous Alma-0 program fragment can be easily translated in C extended with our two functions setChoicePoint and fail. The `SOME` statement is replaced by the call of the `from1ton` function and the boolean expressions are translated into conditions and failures.

```
void main() {
 backTrackInit();
 queens(12);
 fail();
}

int from1ton(int n) {
 int i;
 for(i=1;i<n;i++) {
  if(setChoicePoint()==0) {
   /* set a choice point
      and return i */
   return(i);
  }
 }
 return(i);
}
```

```
queens(int n) {
 int col,row,i;
 for(col=1;col<=n;col++) {
  row=from1ton(n);
  for(i=1;i<col;i++) {
   if(array[i]==row ||
      array[i]==row+col-i ||
      array[i]==row+i-col) {
       /* choose another row */
       fail();
   }
  }
  array[col] = row;
 }
 /* print solution: array[] */
}
```

Remark that the library is initialised by a call to the `backTrackInit` function which creates an initial choice-point in order to prevent using fail before setChoicePoint.

In order to show the potential of the approach, the N-queens solving procedure has been implemented in several languages: standard C, C extended with setChoicePoint and fail, Mercury, Wamcc, Alma-0 and 2LP. In each case, even if it is not the best one, the same searching strategy

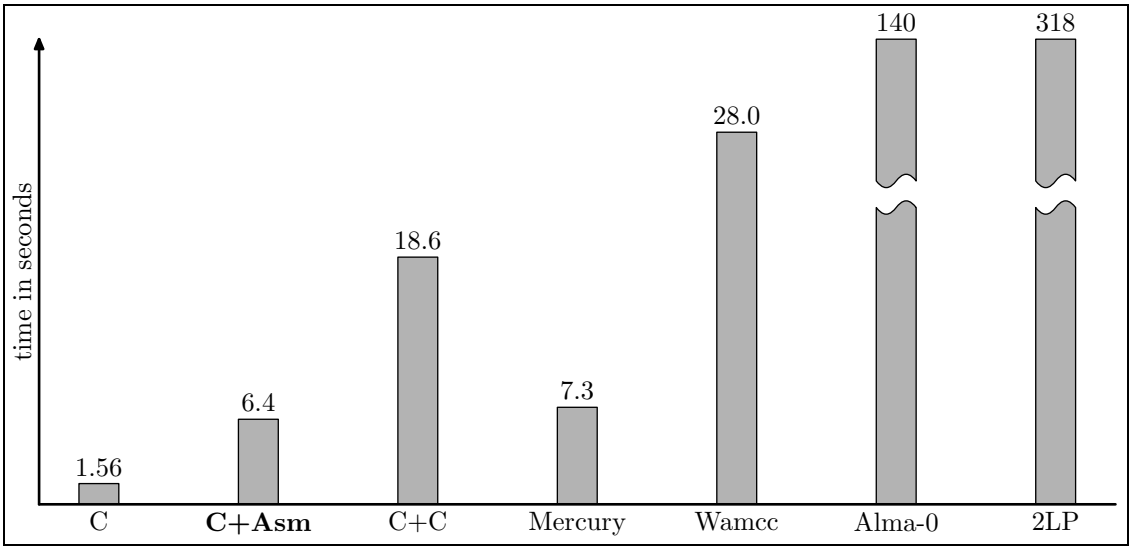


Figure 7.4: Several implementations of the same algorithm are compared in order to show the potential of the approach. The fastest implementation of the N-queens solving procedure is implemented in standard C with a recursive function call. Two others implementations have been done in C extended with setChoicePoint and fail: the first one `C+Asm` use the assembly version of the proposed library and the second one `C+C` use the C version presented in 7.4.1.

has been used: a position for the current queen is chosen and is not removed from the list of remaining positions. The experimental results presented in Figure 7.4 show that the presented approach is as efficient as Mercury. But, you should not deduce that replacing its backtracking management by the proposed library should improve the efficiency. We can only deduce, from these benchmarks, that the proposed approach is a good compromise between simplicity and efficiency.

Let us mention that the use of setChoicePoint and fail is not restricted to the implementation of toy examples such as the N-queens problem: they are intensively used in a more general context related to the compilation of rewriting systems with strategies. In this case, they are frequently used in generated programs that may consist of several thousands lines of C code.

7.6 Concluding Remarks

The previous section has shown that setChoicePoint and fail are designed to extend the C language in order to deal in a natural way with problems that involve search. setChoicePoint and fail can also be used to compile languages with some features inspired by the logic programming paradigm. This is a good alternative to avoid using an abstract machine and to get interesting performances. The two functions were originally designed to compile the ELAN language. Some examples of compilation schemes are given in (Moreau 1998b).

Their "plug-in" designs make them easy to use: conventional programming techniques such as function calls, local variables, parameters passing, modular compilation are compatible with the proposed C language extension. Having a readable code is a key point to be able to design

new complex compilation schemes.

When designing compilation schemes for a new language, the backtracking management, if any, is always a difficult task to solve. The proposed approach seems to be a good compromise between simplicity and efficiency: it provides both high level concepts such as choice points and backtracking, and a low level implementation (in assembly) to get good performances. Experimental results show that the proposed implementation is comparable with an ad-hoc approach like the one used in optimised WAM implementations.

Chapitre 8

Compilation des règles et des stratégies

8.1 Tour d'horizon 113
8.2 Solution retenue pour ELAN 115
8.3 Compilation du filtrage et de la sélection des règles 116
8.4 Compilation des évaluations locales 118
8.5 Construction du terme réduit 124
8.6 Compilation des stratégies 127

Dans le chapitre 4, nous avons défini les grandes lignes du compilateur que nous voulons construire. En particulier, nous avons décidé d'utiliser le langage `C` comme langage cible. Dans les chapitres 5 et 6, nous avons proposé des méthodes de compilation du filtrage syntaxique et du filtrage modulo AC, permettant de dériver des procédures efficaces de normalisation. Dans le chapitre 7, nous avons défini deux primitives de gestion des points de choix, afin de pouvoir compiler plus facilement le double non-déterminisme inhérent au filtrage AC et aux stratégies d'ELAN.

Dans ce chapitre, nous étudions différentes façons d'assembler ces composants et nous proposons de gérer de manière homogène et cohérente la compilation du filtrage, la sélection des règles à appliquer, la compilation des évaluations locales, la compilation des stratégies et la construction du résultat obtenu après application d'une règle ou d'une stratégie.

8.1 Tour d'horizon

Supposons que le langage ELAN ne dispose pas de stratégies définies par l'utilisateur : toutes les règles seraient non nommées. C'est le cas des langages ASF, CafeOBJ, Maude et OBJ-3 présentés dans le chapitre 2. On peut alors imaginer comment compiler ce type de langage : pour un terme clos s représenté en mémoire par une structure de données, il faut sélectionner une ou plusieurs règles $\{l_1 \rightarrow r_1, \ldots, l_n \rightarrow r_n\}$ telles que le membre gauche l_i filtre le sujet s et telles que les éventuelles conditions soient satisfaites. Les méthodes de filtrage many-to-one présentées dans les chapitres 5 et 6 peuvent être utilisées. Il reste alors à sélectionner une règle et à construire le membre droit de la règle en instanciant les variables par une solution du problème de filtrage considéré pour obtenir un nouveau terme clos s'.

L'ordonnancement des quatre étapes *filtrage*, *évaluation de conditions*, *sélection d'une règle* et *construction du terme réduit*, dépend grandement des particularités du langage de spécification et des techniques d'implantation choisies.

Dans l'interpréteur Maude (Clavel et al. 1998) par exemple, le filtrage syntaxique est réalisé par des automates adaptatifs non déterministes, ceci parce que l'aspect réflexif du langage amène le système de réécriture à évoluer dynamiquement : le temps de construction des automates devient alors important. Dans le cas AC, un algorithme de filtrage one-to-one est utilisé pour pouvoir traiter plus efficacement les motifs non-linéaires. Ces choix techniques font que les règles sont *sélectionnées* l'une après l'autre. Pour une règle $l_i \to r_i$ donnée, l'interpréteur cherche un *filtre* de l_i vers s puis *vérifie les conditions* de la règle. Une fois le couple (règle, filtre) trouvé, le *terme réduit* est construit.

Dans Brute (Ishisone et Sawada 1998), qui est une machine abstraite pour CafeOBJ, c'est approximativement la même approche qui a été choisie, mais la stratégie de *sélection* des règles évolue dynamiquement en fonction du terme clos à réduire, ceci pour optimiser l'étape de *filtrage*. Brute intègre une autre optimisation qui consiste à regrouper les règles conditionnelles ayant le même membre gauche, pour factoriser l'étape de filtrage : lorsqu'un filtre correspondant à un ensemble de règles est trouvé, les *conditions sont évaluées* et seulement après, une règle ayant des conditions satisfaisables est *sélectionnée* pour construire le *terme réduit*.

Dans le compilateur ASF+SDF (van den Brand, Klint et Olivier 1999), c'est encore une autre approche qui a été choisie : l'évaluation des *conditions* et la construction du *terme réduit* sont intégrés au processus de *filtrage* many-to-one. Bien qu'ASF+SDF permette de définir des opérateurs associatifs, le filtrage modulo l'associativité est compilé intégralement en des structures de contrôle du langage cible, ce qui n'est pas le cas de l'approche présentée dans le chapitre 6 (des structures de données telles que les graphes bipartis sont engendrées au cours de l'exécution). L'idée consiste à utiliser un automate de filtrage syntaxique et à compiler le filtrage associatif par un ensemble de *boucles* qui énumèrent, de manière exhaustive, les instances possibles des variables apparaissant dans les membres gauches des règles considérées. Après chaque itération, les *conditions* sont vérifiées. En cas d'insatisfaisabilité, l'énumération continue jusqu'à trouver une solution satisfaisante, puis le *terme réduit* est construit, « oubliant » ainsi l'état courant du problème de filtrage considéré. Cette approche est « extrême » dans la mesure où aucune structure de donnée n'est nécessaire au filtrage ; elle est aussi intéressante parce qu'elle ne nécessite aucun mécanisme de gestion du non-déterminisme.

Dans le formalisme ASF, il n'existe aucune construction permettant de remettre en cause l'application d'une règle de réécriture. Cela signifie, que pour une règle et un problème de filtrage donnés, il suffit de trouver une solution satisfaisant les conditions, pour pouvoir appliquer la règle, mais une fois la règle appliquée, il n'est plus nécessaire de se souvenir quelle était la solution.

La situation d'ELAN est, d'une manière générale, différente car certaines règles peuvent être nommées et dans ce cas, leur application est contrôlée par des stratégies éventuellement non-déterministes. C'est pourquoi, dans le cadre d'ELAN, nous ne pouvons pas suivre une approche similaire à celle présentée précédemment. Considérons, par exemple, la règle et la stratégie suivante :

```
rules for Term
  [R1] F(x,y) => g(x,y) end
end
strategies for Term
  []   S1     => dk(R1) end
end
```

où `F` est un symbole AC et `g` un symbole syntaxique. L'application de la stratégie `S1` sur le terme $s =$ `F(a,b)` par exemple, retourne l'ensemble des formes normales atteignables : `g(a,b)`

et `g(b,a)`. Lorsqu'on considère l'application de la règle `R1` sur le terme s, il faut d'une part trouver un filtre de $F(x,y)$ vers $s = F(a,b)$ ($\sigma = \{x \mapsto a, y \mapsto b\}$ par exemple) pour pouvoir appliquer la règle, mais il faut aussi mémoriser l'état courant du problème de filtrage pour pouvoir y revenir (plus tard) et extraire d'autres solutions ($\sigma = \{x \mapsto b, y \mapsto a\}$ par exemple). C'est ce type de situation qui nous empêche de représenter un problème de filtrage AC uniquement par des structures de contrôle. On pourrait naturellement proposer deux schémas de compilation différents suivant qu'il s'agit de règles nommées ou non. Mais les algorithmes de compilation mis en œuvre étant relativement complexes, nous avons opté pour une solution inverse qui consiste à partager au maximum les algorithmes et les structures de données pour compiler aussi bien les règles non nommées que les règles nommées.

8.2 Solution retenue pour ELAN

Lorsqu'on étudie le langage ELAN, et plus précisément son langage de stratégie, on s'aperçoit que seul un sous-ensemble, relativement restreint du langage, a une influence sur l'application des règles. Il faut en fait différencier deux types de constructeurs :

- les constructeurs qui agissent directement sur des règles nommées. Il s'agit de dc one, first one, dc, first et dk lorsqu'ils sont appliqués uniquement à des règles. Leur rôle se restreint alors à spécifier de quelle façon les règles doivent être appliquées : doivent-elles retourner un résultat, tous les résultats correspondant à l'application d'une seule règle, ou tous les résultats correspondant à l'application de toutes les règles ?
- les constructeurs qui agissent directement sur des stratégies. Comme on l'a vu dans le chapitre 1, d'un point de vue théorique il n'y a pas de différence entre une règle de réécriture et une stratégie, mais d'un point de vue pratique il est préférable de séparer ces deux concepts, simplement parce qu'ils se représentent différemment en mémoire. D'un point de vue évaluation, ces constructeurs (dc one/first one/dc/first/dk$(S_1, \ldots, S_n)$, $S_1 ; S_2$, repeat*(S) et iterate*(S)) ne servent qu'à contrôler l'application d'autres stratégies.
  Notons ici que les opérateurs first one et dc one ainsi que les opérateurs first et dc ont des sémantiques différentes mais des implantations identiques. En effet, ce qui différencie dc$(S_1, \ldots, S_n)$ de first$(S_1, \ldots, S_n)$, c'est l'ordre d'application des stratégies : first garantit qu'une stratégie S_i est essayée seulement si les stratégies $S_1, \ldots, S_{i-1}$ échouent (et respectivement pour dc one et first one). Pour des raisons de simplicité, les opérateurs non-déterministes dc one et dc sont implantés par leurs homologues first one et first. Il existe cependant une extension concurrente d'ELAN, étudiée dans (Borovanský et Castro 1998), où ces deux stratégies sont implantées de façon différente, mais l'étude de sa compilation sort du cadre de cette thèse. Dans la suite de ce chapitre, les opérateurs first one et first ne seront plus présentés. Ceci pour des raisons de clarté et de cohérence avec les versions préliminaires d'ELAN, mais il faut cependant les garder à l'esprit.

Lorsqu'une règle est appliquée au cours d'un processus de réécriture, il faut seulement savoir avec quelle stratégie appliquer la règle : dc one, dc ou dk ? Savoir si la règle est appliquée dans le cadre d'une itération, d'une concaténation ou d'une autre construction du langage de stratégie, n'a finalement que peu d'importance. Les deux problèmes peuvent être résolus séparément, mais il faut veiller à ce que la gestion du non-déterminisme des deux approches soit cohérente, afin de rendre plus homogène l'intégration des deux solutions.

Pour des raisons de simplicité et de lisibilité du code généré, nous avons choisi d'utiliser les primitives de gestion de points de choix présentées dans le chapitre 7. La fonction setChoicePoint permet de créer un point de choix en sauvegardant l'environnement d'exécution (les variables

locales du programme C). La fonction fail réactive le dernier point de choix créé et restaure l'environnement d'exécution. Des primitives cutOpen et cutClose permettent de placer des marqueurs dans la pile des points de choix : cutClose détruit tous les points de choix créés depuis le dernier cutOpen. Le contrôle de flots du programme généré est donc principalement géré par ces primitives de gestion du non-déterminisme.

Afin de mieux comprendre la suite de ce chapitre, il est préférable d'avoir une vue d'ensemble de la structure du code généré. Le compilateur ELAN génère un programme composé d'un ensemble de fonctions qui prennent un terme clos s en argument et retournent un nouveau terme clos s' correspondant à l'application d'une règle ou d'une stratégie :

- chaque stratégie S est compilée en une fonction $str_S(s)$;
- chaque ensemble de règles $\{r_1, \ldots, r_n\}$ apparaissant sous un dc one, dc ou dk est compilé en une fonction $rule_r_1 \ldots r_n(s)$;
- l'ensemble des règles non nommées est partitionné en fonction des symboles de têtes des membres gauches des règles. Pour chaque groupe de règles non nommées commençant par un même symbole f, une fonction $fun_f(s)$ est générée.

Lorsqu'aucune règle ne peut s'appliquer sur le terme s, le terme s' ne peut pas être calculé, on dit alors que l'application de la règle ou de la stratégie échoue et un *échec* (fail) est provoqué par la fonction C correspondante. Rappelons qu'un fail réactive le dernier point de choix créé et restaure l'environnement d'exécution. En particulier, lorsqu'un fail est engendré au cours de l'exécution d'une fonction C :

- cela peut réactiver un point de choix posé précédemment dans la fonction ;
- cela peut aussi réactiver un point de choix posé par une autre fonction, dont l'exécution est terminée ou non. Dans ce cas, la fonction courante se termine et l'environnement de la fonction ayant posé le point de choix est restauré.

Si d'un point de vue théorique, l'application d'une stratégie sur un terme retourne un ensemble de résultats, d'un point de vue pratique, l'application d'une stratégie retourne au plus un résultat, et ce sont les points de choix engendrés au cours du calcul qui permettent de mémoriser les contextes intermédiaires. Ce sont les échecs engendrés ultérieurement qui permettent de réactiver une stratégie pour lui faire produire un nouveau résultat. Lorsque tous les résultats sont engendrés, la stratégie produit naturellement un échec.

Pour terminer cette présentation générale, remarquons qu'effectuer une étape de réécriture en appliquant une règle non nommée $r_i \in \{r_1, \ldots, r_n\}$ revient à appliquer la stratégie dc one$(r_1, \ldots, r_n)$ sur le terme s. On comprend alors mieux comment utiliser un même schéma de compilation pour compiler aussi bien les règles non nommées que les règles nommées.

C'est cette volonté d'avoir un code généré lisible, un schéma unique de compilation des règles et une gestion cohérente des points de choix, qui nous a amené à séparer clairement les étapes de *filtrage*, de *sélection d'une règle*, d'*évaluation des conditions* et de *construction du terme réduit*.

Les deux premiers points sont abordés au paragraphe 8.3, et les deux derniers sont respectivement abordés dans les paragraphes 8.4 et 8.5.

8.3 Compilation du filtrage et de la sélection des règles

Considérons un ensemble de règles $\{r_1, \ldots, r_n\}$ et étudions comment générer une fonction C, prenant un terme clos s en argument et retournant un terme s' correspondant à l'application

d'une règle r_i sur s. Afin de respecter les contraintes vues précédemment, cette fonction doit avoir le comportement suivant :

- un fail est engendré lorsqu'aucune règle ne s'applique ;
- si la fonction correspond à un dc one$(r_1, \ldots, r_n)$, une fois le résultat retourné, tous les points de choix posés pendant l'exécution de la fonction sont enlevés en utilisant la primitive cutClose. Ainsi, un échec ultérieur ne réactive pas la fonction, mais réveille un point de choix posé avant l'exécution de la fonction ;
- si la fonction correspond à un dc$(r_1, \ldots, r_n)$, une fois retourné le résultat correspondant à l'application d'une règle r_i, les points de choix ne sont pas enlevés immédiatement, permettant ainsi à la fonction de retourner d'autres résultats lorsqu'un échec réactive un point de choix posé pendant l'évaluation de r_i. C'est seulement après que le dernier résultat calculé, en appliquant r_i, ait été retourné que tous les points de choix posés pendant l'exécution de la fonction sont enlevés en utilisant la primitive cutClose ;
- si la fonction correspond à un dk$(r_1, \ldots, r_n)$, une fois retournés tous les résultats correspondant à l'application d'une règle r_i, la règle r_{i+1} est essayée. Lorsque le dernier résultat calculé, en appliquant r_n, a été retourné, il n'y a alors plus de point de choix posé pendant l'exécution de la fonction qui soit encore actif. Ici encore, un échec ultérieur ne réactive pas la fonction, mais réveille un point de choix posé avant l'exécution de la fonction.

Il devient maintenant possible d'imaginer le schéma de compilation d'un ensemble de règles $\{r_1, \ldots, r_n\}$. Lorsque les membre gauches $l_1, \ldots, l_n$ sont des termes syntaxiques, un automate de filtrage est calculé. Lorsqu'un motif l_i contient un symbole AC, une structure de filtrage AC et des fonctions annexes sont calculées. Le code généré est tel que pour un terme clos s donné, son exécution correspond aux étapes suivantes :

1. la structure de filtrage AC est appliquée sur le terme s et un graphe biparti compact est éventuellement construit. Cette phase détermine l'ensemble des règles $\{r_{i_1}, \ldots, r_{i_m}\}$ qui peuvent potentiellement s'appliquer. Pour connaître l'ensemble des règles qui s'appliquent réellement il reste à vérifier les conditions, et dans le cas AC, il faut en plus extraire et résoudre un graphe biparti ;
2. les règles de $\{r_{i_1}, \ldots, r_{i_m}\}$ sont essayées successivement. Le code de la fonction est donc composé d'une suite de morceaux de programmes qui correspondent respectivement à l'application des règles $r_1, \ldots, r_n$.

   - avant d'évaluer l'application d'une règle r_i on vérifie qu'elle fait bien partie de $\{r_{i_1}, \ldots, r_{i_m}\}$, puis un point de choix est placé pour contrôler l'exécution. Ainsi, tout échec détecté au cours de l'application de r_i, que ce soit au moment du filtrage ou au cours des évaluations locales, permet de revenir à ce point de choix pour y essayer la règle suivante r_{i+1} ;
   - si le membre gauche l_i contient au moins un symbole AC, le graphe biparti correspondant à la règle est extrait et résolu comme décrit dans le chapitre 6. Un point de choix est posé avant chaque solution retournée. Lorsqu'aucune solution n'est trouvée, un échec est provoqué ;
   - les évaluations locales sont ensuite exécutées (voir paragraphe 8.4) et le terme réduit est construit (voir paragraphe 8.5). Il faut juste savoir que toute condition non satisfaite provoque un échec qui peut rendre le contrôle au dernier point de choix posé (évaluation locale précédente, filtrage AC ou règle suivante) ;

– une fois construit le terme réduit, un «saut» permet de passer directement à l'étape 3. Celle-ci est compilée en un morceau de programme se trouvant à la suite des n morceaux correspondant aux $r_1, \ldots, r_n$.

3. cette étape permet de retourner un résultat : le terme réduit. C'est à ce moment là qu'il faut se soucier de la stratégie d'application des $r_1, \ldots, r_n$:

   – si r_i est une règle non nommée, un seul résultat doit être calculé. Il suffit donc de supprimer tous les points de choix posés pendant l'exécution de la fonction en utilisant la primitive cutClose ;

   – si r_i est une règle nommée apparaissant sous un dc one, un seul résultat doit être retourné. Comme précédemment un cutClose est engendré ;

   – si r_i est une règle nommée apparaissant sous un dc, les points de choix posés pendant l'exécution de la fonction ne sont pas supprimés pour permettre d'extraire d'autres solutions : les échecs ultérieurs permettront d'explorer d'autres façons d'appliquer la règle en réactivant des points de choix posés pendant l'étape de filtrage AC ou au cours de l'exécution des évaluations locales.

     Par contre, une fois revenu au point de choix posé avant l'application de r_i, la règle suivante r_{i+1} n'est pas essayée si un résultat a été trouvé ;

   – si r_i est une règle nommée apparaissant sous un dk, aucun point de choix n'est supprimé : on est ainsi sûr que les autres solutions provenant des évaluations locales ou du filtrage AC pourront être extraites. De plus, une fois l'évaluation de r_i terminée, l'étape 2 essayera d'appliquer les règles suivantes et en particulier r_{i+1}. Lorsque r_i est la dernière règle (cas où $i = n$), c'est l'étape 4 qui sera exécutée après son évaluation, pour signaler que tous les résultats ont été extraits.

4. cette étape est exécutée lorsqu'aucune règle de $\{r_1, \ldots, r_n\}$ ne peut s'appliquer, et dans ce cas, un fail est engendré.

Ce schéma général de compilation montre que la solution adoptée pour ELAN est particulière : il y a tout d'abord une étape de présélection qui utilise un automate de filtrage syntaxique. Les règles sont ensuite essayées successivement, et pour chacune d'elles (dont le membre gauche contient un symbole AC) une deuxième étape de filtrage AC (extraction et résolution d'une graphe biparti) est effectuée.

Cette approche hybride est intéressante parce qu'elle permet de sélectionner efficacement une règle, mais elle permet aussi, lorsque c'est nécessaire, d'extraire tous les filtres possibles et de calculer toutes les manières de réduire un terme en utilisant un système de réécriture $\{r_1, \ldots, r_n\}$ donné.

8.4 Compilation des évaluations locales

Comme nous venons de le voir, la compilation d'un ensemble de règles génère une procédure de filtrage many-to-one et pour chaque règle r_i, un morceau de programme ayant la structure

suivante :

r_i : vérification de $r_i \in \{r_{i_1}, \ldots, r_{i_m}\}$
pose d'un point de choix
filtrage AC éventuel
construction d'une substitution
exécution des évaluations locales
construction du terme réduit
saut au morceau de programme correspondant à l'étape 3 de la page 118

Dans ce paragraphe, on s'intéresse principalement à la manière de compiler les évaluations locales apparaissant dans une règle r_i donnée. Comme nous l'avons vu dans le chapitre 1, la structure d'une règle ELAN est la suivante :

```
< règle > ::=
          "[" [ <étiquette> ] "]" <terme> "=>" <terme> { <évaluation locale> }*

< évaluation locale > ::=
          if <terme booléen>
        | where <nom de variable> ":=" "(" [ <stratégie> ] ")" <terme>
        | where "(" <sorte> ")" <terme> ":=" "(" [ <stratégie> ] ")" <terme>
        | choose
            { try { <évaluation locale> }+ }+
          end
```

Ce qui signifie qu'une règle de réécriture peut comporter un nombre quelconque, mais fini, d'évaluations locales. Il faut aussi savoir que l'ordre des évaluations locales est important puisqu'elles sont évaluées dans l'ordre suivant lequel elles apparaissent. Considérons les deux programmes suivants :

```
[] fact(0) => 1                    end
[] fact(1) => 1                    end
[] fact(n) => result
     where result:=() n*fact(n-1)
     if n>1
                                   end
```

```
[] fact(0) => 1                    end
[] fact(1) => 1                    end
[] fact(n) => result
     if n>1
     where result:=() n*fact(n-1)
                                   end
```

Le deuxième système de réécriture est correct alors que le premier ne termine pas toujours. En effet, lorsqu'on évalue la règle :

```
[] fact(n) => result
              where result:=() n*fact(n-1)
              if n>1
   end
```

le calcul d'une forme normale de `n*fact(n-1)` est fait avant de vérifier la condition `n>1`, ce qui peut provoquer une récursion sans fin.

L'exemple suivant est intéressant parce qu'il montre comment implanter en ELAN le problème des 8 reines, et ceci en mélangeant différents styles de programmation. Considérons une signature :

```
module queensAC
import bool int list[int] ;       end
sort   set;                       end
operators global
  queens    :                     list[int];
  ok(@,@,@) : (int int list[int]) bool;
  @ U @     : (set set)           set (AC);
  empty     :                     set;
  @         : (int)               set;
end
```

Considérons un ensemble de règles nommées et une stratégie qui définit un *générateur* : lorsqu'on applique la stratégie `select` à un entier quelconque, on obtient l'ensemble de résultats $\{1, \ldots, 8\}$. Le mécanisme d'évaluation d'ELAN est tel que les entiers sont extraits un à un : la stratégie retourne dans un premier temps l'entier 1, puis la génération d'un échec (fail) déclenche l'extraction de la solution suivante 2, *etc.*

```
rules for int
  x : int;
local
  [r1] x => 1                     end
  [r2] x => 2                     end
  [r3] x => 3                     end
  [r4] x => 4                     end
  [r5] x => 5                     end
  [r6] x => 6                     end
  [r7] x => 7                     end
  [r8] x => 8                     end
end

strategies for int
  [] select => dk(r1,r2,r3,r4,r5,r6,r7,r8) end
end
```

Le système de règles non nommées suivant, permet de vérifier que la position d'une nouvelle reine `d` n'est pas menacée par les positions `p.l` des reines précédemment placées sur l'échiquier. Lorsqu'une nouvelle reine est placée, le prédicat `ok` est appelé avec `diff=1` :

```
rules for bool
  p, d, diff : int;
  l          : list[int];
global
  [] ok(diff,d,nil)     => true                        end
  [] ok(diff,d,p.l)     => false        if d == p      end
  [] ok(diff,d,p.l)     => false        if d-p == diff end
  [] ok(diff,d,p.l)     => false        if p-d == diff end
  [] ok(diff,d,p.l)     => ok(diff+1,d,l)              end
end
```

Enfin, le programme qui permet de calculer une ou l'ensemble des solutions s'écrit avec une seule règle de réécriture. La première version utilise l'aspect non-déterministe du filtrage AC pour extraire des éléments d'un ensemble :

```
rules for list[int]
  p1,p2,p3,p4,p5,p6,p7,p8 : int;
  s1,s2,s3,s4             : set;
local
   [queensrule] queens  => p8.p7.p6.p5.p4.p3.p2.p1.nil
     where (set) p1 U s1 :=() 1 U 2 U 3 U 4 U 5 U 6 U 7 U 8 U empty
     where (set) p2 U s2 :=() s1    if ok(1,p2,p1.nil)
     where (set) p3 U s3 :=() s2    if ok(1,p3,p2.p1.nil)
     where (set) p4 U s4 :=() s3    if ok(1,p4,p3.p2.p1.nil)
     where (set) p5 U s5 :=() s4    if ok(1,p5,p4.p3.p2.p1.nil)
     where (set) p6 U s6 :=() s5    if ok(1,p6,p5.p4.p3.p2.p1.nil)
     where (set) p7 U s7 :=() s6    if ok(1,p7,p6.p5.p4.p3.p2.p1.nil)
     where (set) p8 U s8 :=() s7    if ok(1,p8,p7.p6.p5.p4.p3.p2.p1.nil)
  end
end
```

La deuxième version du programme applique la stratégie `select` sur un terme quelconque (le terme `0` par exemple), pour obtenir des nombres compris entre 1 et 8.

```
   [queensrule] queens  => p8.p7.p6.p5.p4.p3.p2.p1.nil
     where p1:=(select) 0
     where p2:=(select) 0           if ok(1,p2,p1.nil)
     where p3:=(select) 0           if ok(1,p3,p2.p1.nil)
     where p4:=(select) 0           if ok(1,p4,p3.p2.p1.nil)
     where p5:=(select) 0           if ok(1,p5,p4.p3.p2.p1.nil)
     where p6:=(select) 0           if ok(1,p6,p5.p4.p3.p2.p1.nil)
     where p7:=(select) 0           if ok(1,p7,p6.p5.p4.p3.p2.p1.nil)
     where p8:=(select) 0           if ok(1,p8,p7.p6.p5.p4.p3.p2.p1.nil)
  end
```

Ces deux programmes ont volontairement des styles de programmation différents, afin d'illustrer la puissance des évaluations locales et les différents types de non-déterminisme pouvant intervenir au cours de leur exécution. L'application de la règle `queensrule` est contrôlée par l'utilisation de la stratégie `queens` :

```
strategies for list[int]
  [] queens => dk(queensrule) end
end
```

Cette stratégie utilise le constructeur `dk`, ce qui signifie que l'évaluation du programme va retourner un ensemble de résultats correspondant aux différentes façons d'appliquer ces règles. Les résultats correspondent aux 92 solutions du problème des 8 reines.

Essayons dans un premier temps de bien comprendre comment l'application du premier programme permet d'obtenir un résultat. Les évaluations locales sont évaluées dans l'ordre. La première est une condition de filtrage qui consiste à résoudre le problème $p_1 \cup s_1 \leq^?_{AC} 1 \cup \cdots \cup 8 \cup \emptyset$ où p_1 est un entier, s_1 un ensemble et $\emptyset$ est un élément qui symbolise l'ensemble

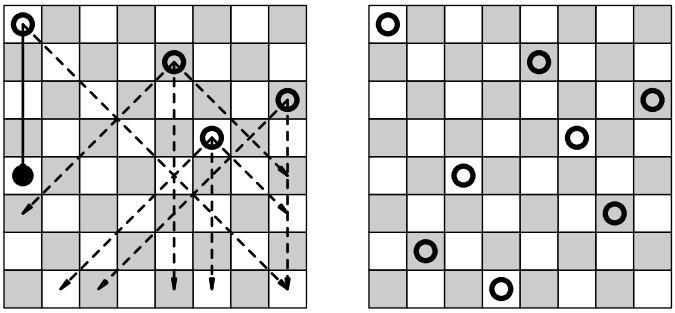


FIG. 8.1 – *La partie gauche de cette figure représente l'état de l'échiquier après y avoir placé 4 reines qui ne se menacent pas. C'est une représentation graphique de la solution partielle* $\{p_1 \mapsto 1, p_2 \mapsto 5, p_3 \mapsto 8, p_4 \mapsto 6\}$*, trouvée au cours de l'exécution des évaluations locales. Le pion noir correspond à une tentative de placer une cinquième reine (*$\{p_5 \mapsto 1\}$*), mais on s'aperçoit qu'elle est mise en échec par la reine se trouvant sur la rangée supérieure de l'échiquier. La partie droite de cette figure est la représentation d'une des 92 solutions qu'il est possible de trouver au problème des 8 reines.*

vide. L'utilisation d'un algorithme de filtrage AC one-to-one permet de trouver une solution, par exemple $\{p_1 \mapsto 1, s_1 \mapsto 2 \cup \cdots \cup 8 \cup \emptyset\}$. La deuxième évaluation locale est aussi une condition de filtrage consistant à résoudre le problème $p_2 \cup s_2 \leq^?_{AC} s_1$. Ici aussi on peut trouver une solution : $\{p_2 \mapsto 2, s_2 \mapsto 3 \cup \cdots \cup 8 \cup \emptyset\}$ par exemple. Une liste `1.nil` est construite puis une condition `ok(1,2,1.nil)` est évaluée pour voir si les deux reines précédemment placées ne sont pas en échec l'une par rapport à l'autre. Étant placées sur une même diagonale, l'évaluation du prédicat `ok` retourne `false` et la condition n'est pas satisfaite, ce qui provoque un fail. Le dernier point de choix, qui avait été posé pendant la résolution de $p_2 \cup s_2 \leq^?_{AC} s_1$ est réactivé et une autre solution du problème de filtrage AC est calculée. Par exemple $\{p_2 \mapsto 3, s_2 \mapsto 2 \cup 4 \cup \cdots \cup 8 \cup \emptyset\}$. L'évaluation reprend alors au niveau de la troisième évaluation locale, et la condition `ok(1,3,1.nil)` est de nouveau évaluée, mais avec succès cette fois. Le processus se poursuit ainsi jusqu'à l'évaluation de la dernière condition.

Considérons maintenant le deuxième programme et supposons que l'évaluation locale courante soit `where p5:=(select) 0`, et que les valeurs 1,5,8 et 6 aient été trouvées pour les variables p_1,p_2,p_3 et p_4. Une représentation graphique de cette solution partielle est donnée sur la figure 8.1.

L'évaluation locale est une condition de filtrage faisant intervenir une stratégie : cela consiste à appliquer la stratégie `select` sur un terme quelconque (ici le terme `0`) et à filtrer la variable `p5` vers le résultat trouvé. Le problème de filtrage est ici trivial. Ce qui est intéressant c'est de bien comprendre comment se déroule l'application de la stratégie `select=dk(r1,...,r8)`. D'après le schéma de compilation présenté au paragraphe 8.3, la fonction `C` correspondant à la stratégie, essaie successivement les règles $r_1, \ldots, r_8$ en posant un point de choix avant chaque application. L'évaluation de `(select) 0` va donc exécuter la fonction `C`, appliquer la règle r_1 et retourner le résultat 1, qui est affecté à `p5`. L'exécution se poursuit par l'évaluation du prédicat `ok(1,1,6.8.5.1.nil)`. Le résultat étant `false` (voir figure 8.1), la condition échoue, un fail est généré, ce qui a pour effet de rendre le contrôle au point de choix posé avant l'application de r_1. Le contexte d'exécution de la fonction est alors réactivé, la règle suivante (r_2) est essayée et la valeur 2 est retournée puis affectée à `p5`.

Ce mécanisme général se poursuit jusqu'à ce que la dernière évaluation locale `if ok(1, p8,p7...p1.nil)` soit évaluée sans échec. Le membre droit de la règle est alors construit : `4.2.7.3.6.8.5.1.nil` est un des 92 résultats que l'on peut obtenir en appliquant la règle `queensrule`.

L'exemple précédent illustre notre manière uniforme de gérer le non-déterminisme lié au filtrage AC, à la sélection d'une règle et à l'évaluation d'une stratégie. Elle consiste à adopter un schéma unique de compilation : lorsque plusieurs possibilités se présentent, un point de choix est posé par setChoicePoint, et lorsqu'un échec se produit, un fail est généré pour explorer les possibilités restantes.

En suivant cette approche, compiler une suite d'évaluations locales revient à les compiler séparément en utilisant les schémas décrits par les algorithmes 8.1 et 8.2.

Algorithme 8.1 Compilation d'une condition : **if** *cond*

1: $c \leftarrow$ évaluation de la condition *cond*
2: **si** $c \neq$ `true` **alors**
3: fail
4: **finsi**

Algorithme 8.2 Compilation d'une condition de filtrage : **where** $p := (S)t$

1: $t' \leftarrow$ un résultat de l'application de la stratégie S sur t
2: filtrage one-to-one de p vers t'
3: **si** filtrage échoue **alors**
4: fail
5: **finsi**

La compilation de la construction **choose** `{` **try** `{` <*évaluation locale*> `}`$^+$ `}`$^+$ est un peu plus complexe : chaque branche **try** `{` <*évaluation locale*> `}`$^+$ est compilée en utilisant les algorithmes 8.1 et 8.2, et des points de choix sont placés entre chaque branche `try ...`, pour permettre, en fonction de la stratégie d'application des règles, de retourner un seul résultat, tous les résultats correspondant à l'exploration d'une branche ou tous les résultats correspondant à l'exploration de toutes les branches. Le schéma de compilation est présenté par l'algorithme 8.3.

Algorithme 8.3 Compilation du `choose/try` : **choose try** $branche_1, \ldots,$ **try** $branche_n$

1: pose d'un point de choix : setChoicePoint
2: compilation de $branche_1$
3: setChoicePoint
4: compilation de $branche_2$
5: ...
6: setChoicePoint
7: compilation de $branche_n$

8.5 Construction du terme réduit

Pour une règle $l \to r$, la phase de construction intervient après le filtrage et une fois que toutes les évaluations locales sont exécutées. On suppose alors que toutes les variables de l ainsi que toutes les variables des motifs des conditions de filtrage sont instanciées par une substitution σ. Il reste à construire le terme clos $r\sigma$ pour pouvoir continuer le processus de normalisation.

La stratégie d'application des règles non nommées étant *leftmost-innermost*, il faudrait, une fois le terme $r\sigma$ construit, rechercher les radicaux les plus internes et les plus à gauche pour les réduire à nouveau. Pour des raisons d'efficacité il est évidemment préférable de ne pas séparer ces trois étapes de construction, recherche, réduction, et de normaliser les sous-termes réductibles pendant la construction de $r\sigma$. Pour cela, les termes sont construits en utilisant un parcours intérieur gauche de l. En fonction du type des nœuds visités, des actions différentes sont effectuées :

- lorsque le nœud correspond à une constante, plutôt que d'allouer de la mémoire à chaque fois, un lien vers un représentant unique de la constante est effectué. Au lancement d'un programme compilé, toutes les constantes apparaissant dans le système de réécriture sont créées en mémoire pour y être partagées par la suite ;
- l'instanciation d'une variable est une opération très simple à réaliser lorsqu'on suppose que la substitution est déjà créée en mémoire : il suffit de créer un lien vers l'instance de la variable correspondante. Dans notre approche, les instances des variables sont référencées par des variables statiques du programme `C` généré. Compiler l'instanciation des variables du membre droit d'une règle revient alors à réutiliser ces variables statiques initialisées pendant l'étape de filtrage ou d'évaluation des conditions de filtrage ;
- lorsque le nœud correspond à un symbole f d'arité n ($n \neq 0$), le problème consiste à construire le terme $t = f(s_1, \ldots, s_n)$ et à calculer sa forme normale. Notons que l'utilisation d'une stratégie *leftmost-innermost* nous assure que les sous-termes $s_1, \ldots, s_n$ sont déjà construits en mémoire et qu'ils sont tous en forme normale.
  - lorsque f est un symbole *constructeur*, le terme t est irréductible et sa construction consiste à allouer de la mémoire pour représenter le symbole f et mémoriser des liens vers les sous-termes $s_1, \ldots, s_n$. Lorsque f est un symbole AC, il faut en plus faire attention à l'ordonnancement des sous-termes s_i : le terme t doit être en forme canonique, ce qui peut nous amener à aplatir et à réordonner certains sous-termes. Le terme $t = f(s_1, \ldots, s_n)$ est construit, de manière incrémentale, en utilisant une fonction mcf qui prend en argument deux termes $t' = f_{AC}(s_1^{\alpha_1}, \ldots, s_p^{\alpha_p})$ et t'' et retourne la forme canonique de $f_{AC}(s_1^{\alpha_1}, \ldots, s_p^{\alpha_p}, t'')$;
  - lorsque f est un symbole *défini*, le terme t est construit mais il est potentiellement réductible parce qu'il existe des règles dont le symbole de tête du membre gauche est f. On utilise alors la fonction fun_f, correspondant à l'ensemble des règles non nommées dont le membre gauche commence par le symbole f, pour essayer de réduire le terme t. Le résultat est soit le terme t (s'il est irréductible), soit une forme normale de t.

Réutilisation du membre gauche. Il existe une méthode bien connue (Sherman 1994, Didrich, Fett, Gerke, Grieskamp et Pepper 1994, Vittek 1996) qui permet d'améliorer l'efficacité du processus de réécriture. L'idée consiste à minimiser le nombre d'allocations mémoire au cours de la construction du terme réduit. Pour cela il est possible d'isoler les constructeurs du sujet, filtrés par le membre gauche de la règle, et de les réutiliser pour construire le terme réduit (*destructive*

update). Un exemple de réutilisation du membre gauche est donné dans la figure 8.2. Cette approche n'est valide que si les constructeurs réutilisés ne sont pas partagés, ce qui oblige à maintenir dynamiquement une information (valeur booléenne ou compteur de références) qui indique pour chaque constructeur, s'il est partagé ou non.

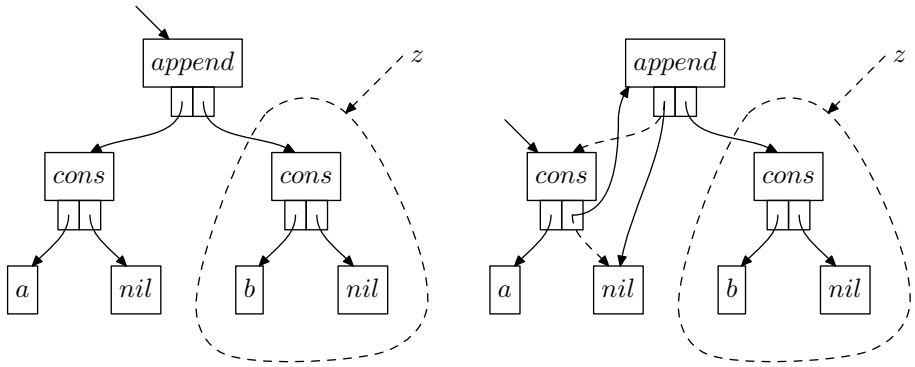


FIG. 8.2 – *Lorsqu'on considère la règle append(cons(e,l),z) → cons(e,append(l,z)) appliquée au terme append(cons(a,nil),cons(b,nil)) par exemple, il est possible de réutiliser les constructeurs append et cons du sujet pour construire le terme réduit cons(a,append(nil,cons(b,nil))). Il suffit alors de modifier deux pointeurs pour éviter toute allocation dynamique de mémoire.*

Dans le cadre de la réécriture avec stratégies, ce type d'optimisation est difficile à mettre en œuvre, simplement parce qu'un retour arrière, provoqué par un fail, peut nécessiter l'accès à une structure qui a pu être réutilisée entre temps. Pour faire cohabiter ce type d'optimisation avec la gestion du non-déterminisme, une méthode consiste à sauvegarder le sujet avant de poser un point de choix. De cette façon, lorsqu'un point de choix est posé, le sujet devient un terme partagé et la réutilisation des constructeurs du membre gauche est inactivée. Dans le cadre d'un calcul déterministe, aucun point de choix n'est posé, ce qui permet de bénificier de l'optimisation proposée.

Construction d'un terme en forme canonique. La construction des graphes bipartis, étudiée dans le chapitre 6, suppose que les motifs et le sujet sont en forme canonique. Plutôt que de construire un terme et de re-calculer sa forme canonique après chaque étape de réécriture, nous proposons de maintenir la forme canonique d'un terme au cours de sa construction. Lorsqu'un nouveau terme t est ajouté comme sous-terme de $s = f_{AC}(s_1^{\alpha_1}, \ldots, s_p^{\alpha_p})$, si un sous-terme s_i équivalent existe déjà, sa multiplicité est incrémentée, sinon, le sous-terme t (qui est en forme canonique par construction) est inséré dans la liste $s_1^{\alpha_1}, \ldots, s_p^{\alpha_p}$ à une position compatible avec l'ordre choisi. Si le symbole racine de t est f_{AC}, une étape d'aplatissement est effectuée et les deux listes de sous-termes sont fusionnées et triées par un algorithme de *merge sort*.

La fonction mcf qui prend en argument deux termes $s = f_{AC}(s_1^{\alpha_1}, \ldots, s_p^{\alpha_p})$ et $t = G(t_1^{\beta_1}, \ldots, t_m^{\beta_m})$ en forme canonique est définie de la manière suivante :

- cas où $f_{AC} \neq G$ (s et t ont des symboles de tête différents)
  - s'il existe i dans $\{1, \ldots, p\}$ tel que $s_i = t$, la multiplicité α_i est incrémentée de un :
    $mcf(f_{AC}(s_1^{\alpha_1}, \ldots, s_p^{\alpha_p}), t) = f_{AC}(s_1^{\alpha_1}, \ldots, s_i^{\alpha_i+1}, \ldots, s_p^{\alpha_p})$
  - sinon, il existe i dans $\{1, \ldots, p\}$ tel que $\forall j \leq i, t < s_j$ et $\forall j > i, s_j < t$:
    $mcf(f_{AC}(s_1^{\alpha_1}, \ldots, s_p^{\alpha_p}), t) = f_{AC}(s_1^{\alpha_1}, \ldots, s_i^{\alpha_i}, t, s_{i+1}^{\alpha_{i+1}}, \ldots, s_p^{\alpha_p})$

– cas où $f_{AC} = G$ (s et t ont les mêmes symboles de tête)
$mcf(f_{AC}(s_1^{\alpha_1}, \ldots, s_p^{\alpha_p}), t) = f_{AC}(u_1^{\gamma_1}, \ldots, u_k^{\gamma_k})$ tel que $(u_1^{\gamma_1}, \ldots, u_k^{\gamma_k})$ est la fusion triée (sans occurrence multiple) de $(s_1^{\alpha_1}, \ldots, s_p^{\alpha_p})$ et $(t_1^{\beta_1}, \ldots, t_m^{\beta_m})$.

De la définition précédente de mcf, il est facile de déduire le résultat suivant : soient $s = f_{AC}(s_1^{\alpha_1}, \ldots, s_p^{\alpha_p})$ et t deux termes en forme canonique, la fonction mcf appliquée à s et t retourne la forme canonique de $f_{AC}(s_1^{\alpha_1}, \ldots, s_p^{\alpha_p}, t)$. En conséquence, la construction d'un terme, en partant des feuilles (*bottom-up*), et en utilisant la fonction mcf, assure que le résultat est en forme canonique.

Renormalisation des instances réductibles. On peut remarquer qu'à chaque fois qu'une fonction C est appelée pour réduire un terme $t = f(s_1, \ldots, s_n)$, les sous-termes $s_1, \ldots, s_n$ sont en forme normale. Et lorsque la règle $l \to r$ est appliquée pour réduire le terme t, on pourrait penser que les instances des variables de l sont elles-même en forme normale. C'est d'ailleurs ce qui nous a amené à réutiliser directement les instances définies par σ pour construire le terme réduit $r\sigma$. C'est en effet le cas, mais seulement pour les variables de l qui n'apparaissent pas directement sous un symbole AC.

Supposons maintenant que la racine du terme t soit un symbole AC. Sa représentation canonique est de la forme $t = f_{AC}(s_1, \ldots, s_n)$. Ici encore, les sous-termes $s_1, \ldots, s_n$ sont irréductibles par construction. Mais lorsqu'on applique une règle de la forme $f_{AC}(x,y) \to r(x,y)$, il se peut que les instances des variables x et y ne soient plus irréductibles : considérons la substitution $\sigma = \{x \mapsto f_{AC}(s_1, \ldots, s_k), y \mapsto f_{AC}(s_{k+1}, \ldots, s_n)\}$, pour $k \geq 2$, l'instance de x peut être réduite par la règle $f_{AC}(x,y) \to r(x,y)$, par exemple. Il faut donc *renormaliser* les instances des variables qui apparaissent directement sous un symbole AC du membre gauche, avant de pouvoir les utiliser pour construire le terme réduit.

La plupart des systèmes étudiés (Maude, OBJ, Brute) n'effectuent pas cette renormalisation au bon moment, ce qui les amène à construire un membre droit de règle réductible. Lorsque $r(x,y)$ est un terme non linéaire en x et que l'instance de x est réductible, ces systèmes construisent une instance de $r(x,y)$ qui risque d'entraîner de multiples renormalisations de x. Il est difficile d'être plus précis ici, simplement parce que le traitement des termes non linéraires dépend grandement des représentations choisies et des optimisations implantées. Dans Maude, par exemple, suivant que les sous-termes réductibles sont partagés ou non, le comportement sera différent. Certaines implantations décorent systématiquement, à l'exécution, tous les termes pour savoir s'ils sont réductibles ou non. Cette approche est assez difficile à mettre en œuvre, et surtout coûteuse en temps. De plus, elle n'aurait que peu d'intérêt pour ELAN, dans la mesure où une grande majorité des termes sont irréductibles par construction (parce que nous utilisons une stratégie de normalisation *leftmost-innermost*).

C'est pourquoi nous avons choisi de renormaliser systématiquement, avant de les utiliser, les instances des variables qui apparaissent directement sous un symbole AC du membre gauche. Les résultats expérimentaux montrent que cette approche permet d'éviter un grand nombre d'étapes de réécriture, simplement parce que le processus de renormalisation est mis en facteur. Nous avons cependant constaté que même si certaines instances de variables étaient potentiellement réductibles, dans la pratique, ces instances sont majoritairement irréductibles. C'est ce qui nous a amené à définir un critère pour déterminer, dans certains cas, l'irréductibilité d'une instance de variable apparaissant directement sous un symbole AC du membre gauche de la règle appliquée.

Soit un motif $l = f_{AC}(x, t_1, \ldots, t_n)$, un sujet $s = f_{AC}(s_1, \ldots, s_m)$ et une substitution σ telle que $l\sigma =_{AC} s$. L'objectif est de définir un critère efficace, pour savoir si $x\sigma$ est irréductible.

On sait que les sous-termes $s_1, \ldots, s_m$ sont en forme normale par construction, on sait aussi

que si l'instance de x est un sous-terme d'un des s_i, elle est elle aussi en forme normale. Partant de ces deux idées de base, nous avons étendu la méthode de construction des termes en « coloriant » certains termes. On distingue alors deux cas :

- à l'exécution, lorsqu'une forme normale $f_{AC}(s'_1, \ldots, s'_{m'})$ est atteinte, tous les sous-termes $s'_1, \ldots, s'_{m'}$ sont coloriés par une même « couleur » ;
- pour construire le terme $f_{AC}(s_1, \ldots, s_m)$, le symbole f_{AC} est construit puis la fonction mcf est utilisée pour ajouter successivement les sous-termes s_i :
  - avant chaque insertion, le sous-terme s_i (qui est irréductible) est colorié par une « couleur » différente de celles associées aux $s_1, \ldots, s_{i-1}$;
  - si l'utilisation de mcf entraîne une étape d'aplatissement et lorsque α sous-termes t identiques apparaissent, ils sont remplacés par une instance unique t^α, mais cette fois, t se voit décoré d'une couleur particulière *bicolore*.

Pour savoir si un terme $x\sigma = f_{AC}(s_1, \ldots, s_k)$ est irréductible, il suffit alors de vérifier que les couleurs des $s_1, \ldots, s_k$ sont bien identiques et qu'aucune d'elles n'est *bicolore*.

Ce critère, très simple à mettre en œuvre, permet de réduire considérablement le nombre de renormalisations inutiles. Les résultats obtenus dans la pratique montrent que ce critère permet de réduire le nombre de renormalisations dans une proportion variant entre 50% et 80%.

8.6 Compilation des stratégies

Dans ce paragraphe, nous nous intéressons à la compilation du langage de stratégie d'ELAN, mais les idées présentées ont un caractère plus général qui peuvent être réutilisées pour compiler tout autre langage de stratégie dont les opérateurs agissent sur les stratégies elles-mêmes.

Les paragraphes 8.3, 8.4 et 8.5 ont montré comment compiler les stratégies élémentaires dc one, dc ou dk$(r_1, \ldots, r_n)$. Étudions maintenant comment compiler les opérateurs agissant sur des stratégies : dc one,dc,dk$(S_1, \ldots, S_n)$, $S_1\,;S_2$, repeat*(S) et iterate*(S)).

Tout comme la compilation d'un ensemble de règles, on peut supposer qu'une stratégie S se compile en une fonction str_S qui prend un terme clos s en argument et retourne un nouveau terme clos s'_i correspondant à l'application de la stratégie. Pour extraire les différents éléments s'_i de l'ensemble des termes $\{s'_1, \ldots, s'_n\}$ atteignables en appliquant la stratégie S au terme s, il suffit d'engendrer un fail pour réactiver un point de choix posé pendant l'exécution de str_S.

Concaténation. Étant données deux stratégies S_1 et S_2, l'opérateur de concaténation $S_1; S_2$ se compile facilement. Il suffit d'enchaîner les fonctions str_S_1 et str_S_2 : la stratégie S_2 est ainsi appliquée aux résultats de S_1. Lorsque S_2 échoue, un nouveau résultat de S_1 est extrait, et lorsque S_1 échoue, la stratégie $S_1; S_2$ échoue également.

Exploration. Étant données n stratégies $S_1, \ldots, S_n$, la compilation de $S =$ dc one,dc ou dk$(S_1, \ldots, S_n)$ s'effectue à l'image de la compilation d'un ensemble de règles, mais sans se soucier de l'étape de filtrage. Suivant l'opérateur appliqué, il faut retourner un seul ou tous les résultats d'une stratégie S_i, ou encore tous les résultats de toutes les stratégies $S_1, \ldots, S_n$.

La compilation de S consiste à compiler chaque sous-stratégie S_i et à essayer d'appliquer successivement les stratégies $S_1, \ldots, S_n$. Le code de la fonction str_S est alors composé d'une suite d'appels aux fonctions $str_S_i(s_i)$ où chaque appel est précédé par la pose d'un point de choix. Lorsqu'un résultat est trouvé pour une sous-stratégie S_i, un saut vers l'étape 1 suivante est effectué :

Compilation du processus de normalisation

```
struct term* normalise_F(struct term *subject ) {
  struct term *res;
  match_state *ms=NULL;
  /* Begin syntactical matching */
  bitSet32_set(mask32,0);
  bitSet32_set(mask32,1);
  /* Begin AC matching */
  indice = MS_init(&ms, match_subterm_F, pattern_list_F);
  }
  ...
  if(bitSet32_get(mask32,1)) {
      struct term *substitution[3];
      /* lhs: F(zExt,f(y,g(b)),f(a,x)) */
      substitution_build(subject,ms,substitution,variable_extract_F,1);
      /* rhs: F(zExt,h(x,y)) */
      if(!isMonoColor(substitution[0])) {
        substitution[0]=normalise_F( substitution[0] );
      }
      TERM_ALLOC(node_h,code_h);
      node_h->subterm[0] = substitution[2];
      node_h->subterm[1] = substitution[1];
      TERM_ALLOC(node_F,code_F);
      term_add_cf_term_color(node_F,substitution[0],color1);
      term_add_cf_term_color(node_F,node_h         ,color2);
      res = normalise_F( node_F );
      goto end;
    } else {
    ...
match_fail:
  res=subject;
end:
  return res;
}
```

Programme 8.1: *Cette figure montre comment une règle comportant un motif AC est compilée : après une étape de filtrage syntaxique (triviale dans cet exemple), le graphe biparti compact est construit par l'intruction :* `MS_init(&ms, match_subterm_F, pattern_list_F)`. *Puis, pour un motif sélectionné ($f_{AC}(z',f(y,g(b)),f(a,x))$ dans cet exemple), la substitution associée à une solution du problème de filtrage est construite par l'instruction* `substitution_build(subject, ms, substitution, variable_extract_F, 1)`. *Comme décrit dans ce paragraphe, un test (*`if(!isMonoColor(substitution[0]))`*) est effectué pour déterminer si l'instance est réductible ou non. La dernière partie de cette fonction consiste à construire le terme réduit et à calculer sa forme canonique en utilisant* `term_add_cf_term_color`. *Cette dernière fonction correspond à une implantation de la fonction mcf où il est possible de donner une couleur aux termes insérés sous le symbole AC.*

1. tout comme dans l'algorithme présenté au paragraphe 8.3, cette étape permet de retourner un résultat, et c'est à ce moment précis qu'il faut se soucier de la stratégie d'application des $S_1, \ldots, S_n$.
   - si l'opérateur est un dc one : un seul résultat doit être retourné. Il suffit donc de supprimer tous les points de choix posés pendant l'exécution de la fonction str_S en utilisant la primitive cutClose ;
   - si l'opérateur est un dc : tous les résultats associés à S_i doivent pouvoir être retournés. Il suffit de ne pas supprimer les points de choix posés pendant l'exécution de str_S_i, pour permettre d'extraire d'autres solutions. Par contre, une fois revenu au point de choix posé avant l'exécution de str_S_i, la stratégie suivante S_{i+1} n'est pas essayée si un résultat a été trouvé ;
   - lorsque l'opérateur est un dk, aucun point de choix n'est supprimé : on est ainsi sûr que toutes les solutions pourront être extraites. De plus, une fois l'exécution de str_S_i terminée, la stratégie suivante S_{i+1} est essayée.
2. cette étape est exécutée lorsqu'aucune stratégie de $\{S_1, \ldots, S_n\}$ ne peut s'appliquer, et dans ce cas, un fail est engendré.

Répétition. La stratégie iterate*(S) applique répétitivement la stratégie S et retourne tous les résultats intermédiaires des applications successives de S. L'itération se termine lorsque pour un terme donné, la stratégie S ne peut plus s'appliquer.

Le schéma de compilation est relativement simple puisqu'il consiste à générer une boucle dans laquelle un point de choix est placé avant toute exécution de la stratégie S (voir algorithme 8.4).

Algorithme 8.4 iterate*(S)

```
boucler
  si setChoicePoint=0 alors
    break
  finsi
  code correspondant à S
fin boucle
```

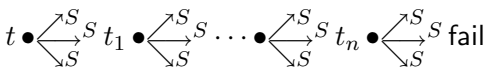
Cette stratégie est particulière dans la mesure où elle n'échoue jamais : zéro application de S est possible.

La stratégie repeat*(S) est semblable à iterate*(S), mais seuls les résultats correspondant aux dernières applications de S sont retournés. Le schéma de compilation est un peu plus complexe que celui d'iterate*(S) (voir algorithme 8.5). En effet, tout comme dans l'algorithme 8.4, un point de choix doit être posé lors de chaque itération, mais la différence vient du fait que tous les résultats correspondant à l'application d'une itération de S ne peuvent pas être extraits au fur et à mesure : l'exploration doit se faire en profondeur d'abord (pour retourner les feuilles) et non en largeur d'abord, comme c'était le cas d'iterate*(S). L'idée consiste à utiliser un marqueur *succes* qui est positionné à $\top$ lorsque S peut s'appliquer. Lorsque S échoue, l'avant-dernière application de S redevient active pour pouvoir extraire les éventuelles autres solutions. Mais une fois que toutes les solutions sont extraites, le marqueur est inspecté pour savoir si l'on doit continuer ou revenir à l'antépénultième application de S : lorsque le marqueur est positionné à $\bot$, il ne faut évidemment pas continuer, sous peine de réessayer la stratégie S qui vient d'échouer.

Algorithme 8.5 repeat*(S)

```
boucler
  succes ← ⊥
  si setChoicePoint=0 alors
    break
  sinon si succes = ⊤ alors
    fail
  finsi
  code correspondant à S
  succes ← ⊤
fin boucle
```

Partant d'un terme t, on s'aperçoit que l'application de repeat*(S) entraîne la pose d'un point de choix à chaque itération :



Ce qui signifie qu'en plus des points de choix posés par les applications de S, il y a autant de points de choix, actifs simultanément, que la longueur de la répétition. Ce qui peut poser des problèmes évidents de gestion mémoire. Mais il semble que ce soit le prix à payer pour bénéficier de la puissance d'un tel constructeur de stratégie.

Dans le chapitre 9, nous verrons comment une analyse fine des règles et des stratégies composant un programme peut permettre de générer des schémas de compilation permettant de rendre plus efficace, en temps et en mémoire, l'exécution d'une telle stratégie, en ne posant qu'un seul point de choix.

Chapitre 9

Analyse du déterminisme

9.1 Stratégies primitives 132
9.2 Classification du déterminisme 132
9.3 Inférence de la classe de déterminisme 134
9.4 Impact de l'analyse du déterminisme 136
9.5 Résultats expérimentaux 138

Le non-déterminisme est une notion inhérente au processus de réécriture. En effet, la possibilité d'appliquer simultanément plusieurs règles sur un même terme et la possibilité de définir des systèmes de réécriture non confluents nous amènent à considérer qu'un terme peut avoir plusieurs formes normales, s'il en existe. Il faut aussi noter que la présence de symboles AC dans une signature est une source supplémentaire de non-déterminisme, parce qu'une règle peut s'appliquer de plusieurs façons possibles.

Pour prendre en compte ces ensembles de résultats, nous avons introduit le concept de stratégie : une stratégie est une fonction qui retourne un ensemble de résultats lorsqu'elle est appliquée sur un terme initial. D'un point de vue pratique, ces ensembles de résultats ne sont pas représentés explicitement mais leurs éléments peuvent être énumérés grâce à un mécanisme de gestion de points de choix (voir chapitre 7).

ELAN n'est évidemment pas le seul langage à intégrer des constructions non déterministes pour gérer des ensembles de résultats. On peut ainsi citer les langages de la famille Prolog tels que Wamcc (Diaz 1995) ou Mercury (Henderson, Conway et Somogyi 1996), ou encore des langages qui mélangent les paradigmes de programmation impérative avec ceux de la programmation logique : Claire (Caseau et Laburthe 1996), Alma-0 (Partington 1997, Apt et Schaerf 1997) ou 2LP (McAloon et Tretkoff 1995), par exemple.

L'étude du langage de stratégie et de la réécriture modulo AC nous a dans un premier temps amené à développer des algorithmes généraux de compilation qui traitent de manière uniforme le non-déterminisme inhérent aux stratégies et au filtrage AC (voir chapitre 8). S'inspirant de (Henderson, Somogyi et Conway 1996), nous proposons, dans ce chapitre et dans (Kirchner et Moreau 1998), un algorithme permettant de déterminer si une stratégie ou un ensemble de règles a un comportement non-déterministe (*i.e.* si on peut obtenir plusieurs résultats).

9.1 Stratégies primitives

Le comportement non-déterministe d'ELAN est essentiellement dû aux constructeurs de stratégies dc one, dc et dk qui permettent de spécifier de quelle manière un ensemble de règles ou de stratégies doit être appliqué. Plus généralement, deux notions se cachent derrière ces opérateurs :

- la sélection d'une ou plusieurs règles ou stratégies à appliquer ;
- pour une règle ou une stratégie donnée, la sélection d'un ou plusieurs résultats liés à son application.

Afin de proposer un algorithme d'analyse du déterminisme qui ne soit pas restreint au cadre d'ELAN, nous proposons d'introduire quatre opérateurs élémentaires.

Contrôle de la sélection. Étant donnés un terme clos t et un ensemble de stratégies $S = \{S_1, \ldots, S_n\}$, $n \geq 1$:

- l'opérateur select one sélectionne une stratégie $S_i \in S$ telle que l'application de S_i à t n'échoue pas ;
- l'opérateur select all sélectionne le plus grand sous-ensemble $S' \subseteq S$ tel que $\forall S_i \in S'$, l'application de S_i à t n'échoue pas.

Dans les deux cas, l'opérateur échoue si l'ensemble des résultats de toutes les stratégies est vide.

Contrôle du nombre de résultats. Étant donnée une stratégie S :

- l'opérateur one construit une stratégie $\mathsf{one}(S)$ qui retourne au plus un résultat parmi ceux de l'application de S à un terme t quelconque ;
- l'opérateur all construit une stratégie $\mathsf{all}(S)$ qui retourne tous les résultats correspondant à l'application de S à un terme t quelconque.

En utilisant ces quatre primitives, les constructeurs de stratégie d'ELAN sont définis par les axiomes suivants, où S_i est aussi bien une stratégie qu'une règle de réécriture :

$$\begin{array}{lcl} \mathsf{dc\ one}(S_1, \ldots, S_n) & = & \mathsf{select\ one}(\mathsf{one}(S_1), \ldots, \mathsf{one}(S_n)) \\ \mathsf{dc}(S_1, \ldots, S_n) & = & \mathsf{select\ one}(\mathsf{all}(S_1), \ldots, \mathsf{all}(S_n)) \\ \mathsf{dk}(S_1, \ldots, S_n) & = & \mathsf{select\ all}(\mathsf{all}(S_1), \ldots, \mathsf{all}(S_n)) \end{array}$$

Notons que les opérateurs dc et dk sont équivalents lorsqu'ils sont appliqués à un argument unique : $\mathsf{dc}(S) = \mathsf{dk}(S) = S$.

9.2 Classification du déterminisme

Nous classifions les stratégies en cinq catégories, en fonction du nombre maximum de résultats qu'elles permettent de calculer (un ou plus de un), et suivant qu'elles peuvent échouer ou non. Dans ce paragraphe, nous ne considérons que des termes t et des stratégies S telles que l'application de S à t termine. Lorsqu'il s'agit d'évaluer le nombre de résultats qu'il est possible d'obtenir en appliquant la stratégie $\mathsf{dk}(a \to b)$ sur un terme t quelconque, il est naturel de considérer que l'application de la stratégie peut échouer (si $t \neq a$ la règle ne peut pas s'appliquer) ou retourner un seul résultat (si $t = a$). Par extension, nous disons que l'application de la stratégie $\mathsf{dk}(a \to a)$ retourne au plus un résultat et que l'application de $\mathsf{repeat*}(\mathsf{dk}(a \to a))$ sur un terme t quelconque retourne « un et un seul résultat », même s'il est clair que l'application de cette stratégie au terme $t = a$ ne termine pas. Il faut ainsi considérer l'expression que « un et un seul

résultat » signifie « lorsque l'application de la stratégie termine, nous ne pouvons obtenir qu'un seul résultat ».

En adoptant la même terminologie que celle présentée dans (Henderson, Somogyi et Conway 1996), nous obtenons la classification suivante :

- une stratégie S est dite *déterministe* (det) si pour tout terme t, son application $S(t)$ retourne exactement un résultat ;
- une stratégie S est dite *semi-déterministe* (semi) si pour un terme t quelconque, son application $S(t)$ échoue ou retourne au plus un résultat ;
- une stratégie S est dite *multi-résultats* (multi) si pour un terme t quelconque, son application $S(t)$ n'échoue jamais et retourne au moins un résultat ;
- une stratégie S est dite *non-déterministe* (nondet) si pour un terme t quelconque, son application $S(t)$ échoue ou retourne un ou plusieurs résultats ;
- enfin, une stratégie qui échoue tout le temps est dite d'*échec* (fail).

Ces différentes catégories définissent des *modes* (d-mode) et un ordre partiel peut être établi comme suit :

$$\mathsf{det} < \mathsf{semi},\ \mathsf{multi} < \mathsf{nondet}$$

Cet ordre correspond intuitivement à la notion d'inclusion sur les intervalles dont les bornes sont le nombre minimal et maximal de résultats qu'il est possible d'obtenir :

$$[1,1] < [0,1],\ [1,n] < [0,n]$$

où n est un entier arbitraire strictement supérieur à 1.

Au paragraphe 9.3, nous proposons un algorithme qui permet d'inférer le mode d'une stratégie particulière. Pour cela, deux opérateurs commutatifs *And* et *Or*, définis sur une logique à cinq valeurs, sont nécessaires. Leur définition est donnée ci-dessous. L'intuition qui se cache derrière ces opérateurs est la suivante :

- *And* permet de calculer le mode correspondant à la composition de deux stratégies S_1 et S_2. Lorsque ce mode est semi-déterministe par exemple, cela signifie qu'une des deux stratégies S_1 ou S_2 peut échouer et qu'aucune des deux stratégies ne peut retourner plus d'un résultat ($And(\mathsf{det},\mathsf{semi}) = And(\mathsf{semi},\mathsf{det}) = And(\mathsf{semi},\mathsf{semi}) = \mathsf{semi}$) ;
- *Or* permet de calculer le mode correspondant à l'application concurrente de deux stratégies : S_1 ou S_2. Cela permet de caractériser le nombre de résultats qui composent l'union des résultats de S_1 et de S_2.

And	det	semi	multi	nondet	fail
det	det	semi	multi	nondet	fail
semi	semi	semi	nondet	nondet	fail
multi	multi	nondet	multi	nondet	fail
nondet	nondet	nondet	nondet	nondet	fail
fail	fail	fail	fail	fail	fail

Or	det	semi	multi	nondet	fail
det	multi	multi	multi	multi	det
semi	multi	nondet	multi	nondet	semi
multi	multi	multi	multi	multi	multi
nondet	multi	nondet	multi	nondet	nondet
fail	det	semi	multi	nondet	fail

9.3 Inférence de la classe de déterminisme

L'algorithme d'inférence du mode de déterminisme est présenté en trois étapes : pour une stratégie, il utilise la forme décomposée en stratégies primitives. Pour une règle de réécriture, les évaluations locales sont analysées. Enfin, l'algorithme traite le problème de récursivité, dû à la possibilité de créer un cycle de dépendance entre les règles et les stratégies, en associant un mode particulier.

Inférence du d-mode d'une stratégie

Le d-mode d'une stratégie correspond à son type de déterminisme. Celui-ci est inféré à partir de son expression sous forme de stratégies primitives (one, all, select one et select all).

- d-mode(one(S)) = semi si S est une règle de réécriture (le filtrage peut échouer). Sinon, on a :
  $$\text{d-mode(one}(S)) = \begin{cases} \text{det} & \text{si d-mode(S) est det ou multi} \\ \text{semi} & \text{si d-mode(S) est semi ou nondet} \end{cases}$$
- d-mode(all(S)) = And(semi,d-mode(S)) si S est une règle de réécriture (le filtrage peut échouer). Autrement, d-mode(all(S)) = d-mode(S)
- $$\text{d-mode(repeat*(S))} = \begin{cases} \text{det} & \text{si d-mode(S) est det ou semi} \\ \text{multi} & \text{si d-mode(S) est multi ou nondet} \end{cases}$$
  L'opérateur repeat* ne peut pas échouer, simplement parce que zéro itération est toujours possible. Remarquons alors que si S n'échoue jamais, l'application de la stratégie repeat*(S) ne termine pas et ne retourne aucun résultat.
- d-mode(iterate*(S)) = multi. L'opérateur iterate* ne peut pas non plus échouer. En général, il retourne plusieurs résultats parce que toutes les étapes de l'itération sont considérées comme des résultats. De même que repeat*, si S n'échoue jamais, l'application de iterate*(S) ne termine pas, mais cela peut être utile, dans certains cas, pour représenter des générateurs ou des structures de données infinies (un résultat est retourné à chaque itération).

- d-mode($S_1; S_2$) = And(d-mode(S_1),d-mode(S_2)).
- d-mode(select one($S_1, \ldots, S_n$)) = And(d-mode(S_1), ... ,d-mode(S_n))
- d-mode(select all($S_1, \ldots, S_n$)) = Or(d-mode(S_1), ... ,d-mode(S_n))

Inférence du d-mode d'une règle

Pour calculer le d-mode d'une règle de réécriture il suffit d'analyser les d-mode des évaluations locales :

- Commençons par considérer le cas d'une condition « simple » **if** c ou d'une condition de filtrage **where** $p := ()c$ ne faisant pas intervenir de stratégie. Le calcul de c', la forme normale du terme c (par rapport aux règles non nommées) ne peut pas échouer. Si $c' \neq \top$ ou si p ne filtre pas c', la condition échoue et la règle risque de ne pas s'appliquer, mais cela ne modifie pas le nombre maximum de résultats qu'il est possible d'obtenir en appliquant la règle (on sait déjà que la borne inférieure est 0 puisque le filtrage de la règle peut échouer). Le d-mode d'une telle évaluation locale est donc det (c'est un élément neutre pour l'opérateur And).
  La seule situation permettant à la règle de retourner plusieurs résultats, se produit lorsqu'une variable de c apparaît sous un symbole AC du membre gauche de la règle ou sous un symbole AC d'un motif d'une condition de filtrage précédente. Dans ce cas, l'évaluation locale est dite multi-résultats (multi).
- Considérons maintenant une condition de filtrage **where** $p := (S)c$ qui implique l'application d'une stratégie S. L'évaluation locale a dans ce cas le même d-mode que celui de la stratégie S. Cependant, comme dans le cas précédent, lorsqu'une variable de c apparaît sous un symbole AC du membre gauche de la règle ou d'un motif d'une condition de filtrage précédente, le d-mode de l'évaluation locale courante est soit multi, soit nondet et se calcule par And(multi,d-mode(S)).
- Lorsque l'évaluation locale est un `choose try ... end`, il faut la voir comme un moyen de mettre en facteur un ensemble de règles ayant un même membre gauche. Le d-mode de l'évaluation locale dépend donc de la stratégie d'application de la règle : si l'opérateur d'application est un one, il faut calculer la conjonction des d-mode de chaque branche `try ...` avec l'opérateur And. Si l'opérateur de stratégie est un all, il faut calculer la disjonction des d-mode de chaque branche `try ...` avec l'opérateur Or. Le d-mode d'une branche est la conjonction du d-mode des sous-évaluations locales la composant.

Le d-mode d'une règle R se calcule en effectuant la conjonction (opérateur And) des modes de ses évaluations locales. Lorsque la règle ne possède aucune évaluation locale, son mode est dit déterministe : d-mode=det. L'application d'une règle peut évidemment échouer, mais le type d'échec n'est pas le même suivant qu'il s'agisse d'une règle nommée ou non :

- lorsqu'une règle non nommée ne peut pas s'appliquer sur un terme t, le terme n'est pas modifié. D'un point de vue analyse du déterminisme, on a bien un et un seul résultat, et c'est pourquoi une règle sans évaluation locale est dite déterministe ;
- lorsqu'une règle nommée ne peut pas s'appliquer sur un terme t, un échec (fail) est engendré et aucun résultat n'est obtenu. La règle est toujours déterministe, mais sa stratégie d'application ne l'est plus : d-mode(one(S)) = semi si S est une règle de réécriture.

Problème lié à la récursivité

La définition d'une règle ou d'une stratégie peut dépendre, d'une manière générale, de stratégies impliquant cette même règle ou stratégie. Le calcul d'un d-mode particulier peut ainsi dépendre de lui-même. Un problème similaire arrive en programmation logique, lorsqu'il s'agit de définir le mode d'un prédicat (Sawamura et Takeshima 1985).

Pour éviter la non-terminaison de notre algorithme d'analyse du déterminisme, lorsqu'un d-mode dépend de lui-même, un mode par défaut est donné. Pour une primitive de stratégie donnée, ce mode correspond à son mode maximum (en utilisant l'ordre défini au paragraphe 9.2) :

primitive	one	all	repeat*	iterate*	;
d-mode *par défaut*	semi	nondet	multi	multi	nondet

9.4 Impact de l'analyse du déterminisme

Connaître au moment de la compilation le d-mode d'une règle ou d'une stratégie a un impact considérable sur la qualité du code généré. Il devient en effet possible de générer un code particulier pour les stratégies déterministes ou semi-déterministes, et par la même occasion, la pose d'un grand nombre de points de choix peut être évitée. Les améliorations apportées se constatent non seulement au niveau des performances du programme généré mais aussi en terme d'espace mémoire nécessaire pour produire des résultats. Et dans de nombreux cas, l'analyse du déterminisme a permis de faire terminer correctement des programmes qui s'arrêtaient à la suite d'un manque de mémoire.

Nous présentons dans ce paragraphe les différents composants du compilateur qui peuvent tirer un bénéfice de cette phase d'analyse du déterminisme.

Détection d'erreurs. Comme mentionné au paragraphe 9.3, l'analyse du déterminisme peut aider à détecter, au cours de la compilation, la non terminaison des stratégies du type repeat*(S) ou iterate*(S), lorsque le d-mode de S est det ou multi. Cette remarque peut paraître anodine, et pourtant, on sait bien que bon nombre des erreurs de programmation viennent d'une mauvaise re-combinaison de modules « indépendants ». Ici, il faut voir S comme une stratégie extraite d'une bibliothèque de stratégies, et il ne devient plus évident de savoir si sa combinaison avec l'opérateur repeat*, construit une stratégie qui termine ou non.

Dans Mercury par exemple, le langage impose aux programmeurs de définir le d-mode d'un prédicat au moment de sa définition. Les concepteurs du langage affirment que, tout comme le typage des variables, cela permet de réduire considérablement les risques d'erreurs.

Filtrage AC. Au paragraphe 6.6, nous avons présenté un algorithme glouton permettant d'améliorer l'efficacité du filtrage AC dans le cas de règles non conditionnelles ou dont les conditions ne dépendent pas de variables apparaissant sous un symbole AC du membre gauche. C'est en appliquant l'analyse du déterminisme que les règles gloutonnes sont sélectionnées : celles dont le d-mode est det ou semi.

Sélection et application d'une règle. Au paragraphe 8.3, nous avons présenté un schéma général de compilation des règles où un point de choix était placé avant chaque application d'une règle r_i. Au paragraphe 8.4, nous avons présenté un schéma général de compilation des évaluations locales où un point de choix était aussi placé avant chaque évaluation d'une condition de filtrage. Et c'est

dans l'étape 3 du schéma de compilation des règles (voir page 118), qu'un traitement particulier est effectué pour enlever ces points de choix, en fonction de la stratégie d'application de la règle.

L'analyse du déterminisme permet d'agir à trois niveaux :

- lorsqu'une règle ne contient que des évaluations locales déterministes (det) ou semi-déterministes (semi), il n'est plus nécessaire de placer un point de choix entre chaque évaluation locale : tout échec implique l'impossibilité d'appliquer la règle considérée ;
- lorsqu'un ensemble de règles $\{r_1, \ldots, r_n\}$ ne contient que des règles déterministes (det), cela signifie que toutes les évaluations locales sont det. On sait alors qu'un échec ne pourra provenir que d'une condition insatisfaite, et non d'une stratégie, puisqu'une stratégie déterministe ne peut pas échouer. Le schéma de compilation d'un ensemble de règles peut alors être modifié pour ne plus engendrer de fail et générer un « saut » vers l'évaluation de la règle suivante en cas d'échec d'une condition. Ainsi, il n'est plus nécessaire de placer un point de choix avant chaque application de règle ;
- lorsqu'une stratégie dc one, dc ou dk$(r_1, \ldots, r_n)$ est semi-déterministe (elle ne peut pas être det, parce que le filtrage peut toujours échouer), on sait qu'un seul résultat doit être calculé. Il devient donc possible de modifier l'étape 3 (de la page 118) pour que tous les points de choix, posés pendant l'évaluation de la stratégie, soient supprimés. La génération des cutClose, ne se fait donc plus en fonction d'un critère syntaxique (présence ou non d'un dc one) mais en fonction du d-mode (semi-déterministe ou non).
  Cette remarque s'applique aussi à l'étape 1 de l'algorithme de compilation des stratégies dc one, dc ou dk$(S_1, \ldots, S_n)$, présentée page 127.

L'intégration de ces optimisations, dans le compilateur, a un impact important sur la vitesse d'exécution des programmes générés. Elles permettent en effet de réduire considérablement le nombre de points de choix posés dynamiquement, ce qui diminue d'autant le temps passé dans la gestion du non-déterminisme.

Compilation des stratégies. Au paragraphe 8.6, nous avons présenté un schéma de compilation de la stratégie repeat*(S) tel que l'exploration se fait en profondeur d'abord :

$$t \bullet \overset{S}{\underset{S}{\nearrow\!\searrow}} S\ t_1 \bullet \overset{S}{\underset{S}{\nearrow\!\searrow}} S \cdots \bullet \overset{S}{\underset{S}{\nearrow\!\searrow}} S\ t_n \bullet \overset{S}{\underset{S}{\nearrow\!\searrow}} S\ \mathsf{fail}$$

La pose d'un point de choix à chaque étape permet de « marquer » les étapes de l'itération et de savoir lorsqu'une nouvelle voie doit être explorée.

Mais lorsque la stratégie S est det ou semi, la question de savoir si une nouvelle voie doit être explorée, ne se pose plus. Il suffit d'appliquer continuellement S, de mémoriser le résultat intermédiaire à chaque étape, et de le retourner lorsque l'application de S échoue :

$$\bullet\, t \longrightarrow^S t_1 \longrightarrow^S \cdots \longrightarrow^S t_n \longrightarrow \mathsf{fail}$$

On peut ainsi définir un nouveau schéma de compilation, présenté dans l'algorithme 9.1, qui ne pose plus qu'un seul point de choix, indépendamment de la longueur de l'itération.

Cette optimisation influence naturellement le temps d'exécution, mais son principal apport est de réduire considérablement l'espace nécessaire pour exécuter une itération : le nombre de points de choix actifs simultanément, qui était égal à la longueur de l'itération, est maintenant réduit à 1. Sachant que le calcul d'une forme normale d'un terme est essentiellement l'application répétitive d'une stratégie, il est fréquent d'effectuer des milliers, voire des millions d'itérations. On imagine alors facilement, que l'absence d'une telle optimisation pouvait poser des problèmes de gestion mémoire lorsqu'il fallait mémoriser plusieurs milliers d'environnements, simultanément.

Algorithme 9.1 repeat*(S)

```
lastTerm ← sujet
si setChoicePoint=0 alors
  boucler
    lastTerm ← valeur retournée par l'application de S
  fin boucle
finsi
```

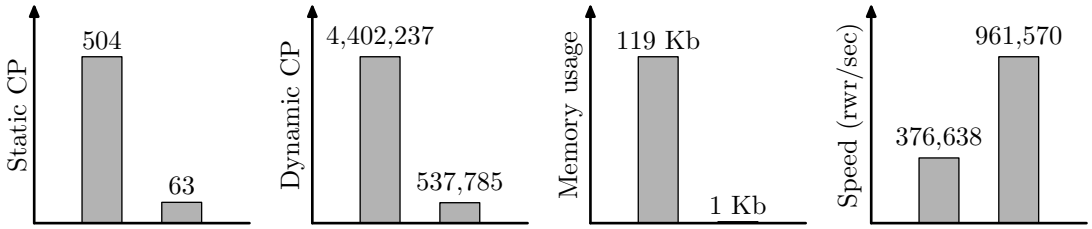
Construction du terme réduit. Au paragraphe 8.5, nous avons présenté une optimisation permettant de réduire le nombre d'allocations mémoire en réutilisant des morceaux du membre gauche pour constuire le terme réduit. Cette optimisation ne peut malheureusement pas s'appliquer lorsque des points de choix sont posés, parce que les termes deviennent partagés. L'analyse du déterminisme permet d'une part de réduire le nombre de points de choix posés et d'autre part de déterminer des séquences de calcul pendant lesquelles aucun point de choix n'est posé. Ces deux informations permettent ainsi de générer du code plus efficace pour construire les termes réduits associés à des règles ou à des stratégies déterministes.

9.5 Résultats expérimentaux

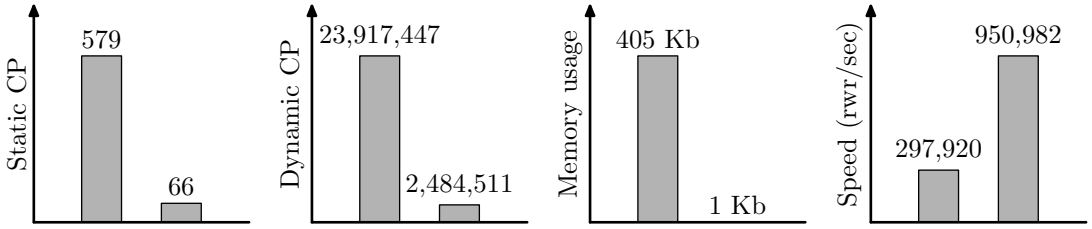
Dans ce paragraphe, nous proposons d'observer l'impact de l'analyse du déterminisme d'un point de vue expérimental. Nous avons pour cela sélectionné des programmes de différents domaines. Chaque programme est compilé et exécuté deux fois : une première fois sans aucune optimisation liée à l'analyse du déterminisme, et une deuxième fois avec l'analyse du déterminisme activée.

Les résultats sont présentés sous forme d'histogrammes. Pour chaque programme nous donnons le nombre d'instructions setChoicePoint générées par le compilateur (Static CP), le nombre de points de choix créés au cours de l'exécution (Dynamic CP), la mémoire utilisée pour mémoriser les environnements (Memory usage) et le nombre de règles de réécriture appliquées par seconde (rwr/sec). Les mesures ont été faite sur une station Dec Alpha.

– les programmes p5 et p8 correspondent à la complétion de Knuth-Bendix appliquée à des versions modifiées de la théorie des groupes. Les modifications consistent à introduire 5 (respectivement 8) éléments neutres, ainsi que 5 (respectivement 8) éléments inverses. Ces théories sont des tests fréquemment utilisés pour évaluer les performances des prouveurs automatiques de théorèmes. L'exécution de p5 donne les résultats suivants :

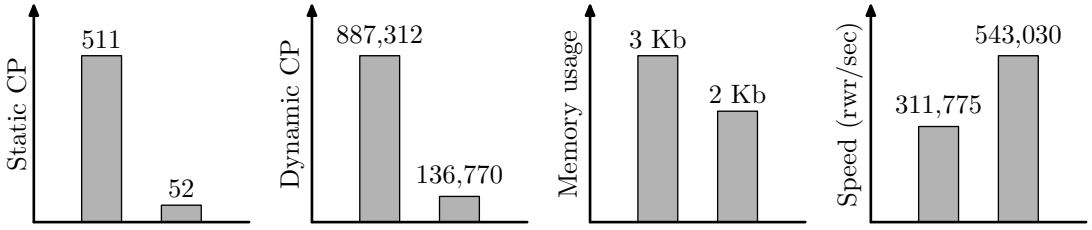


L'exécution de p8 utilise approximativement le même programme, mais implique des calculs plus complexes :



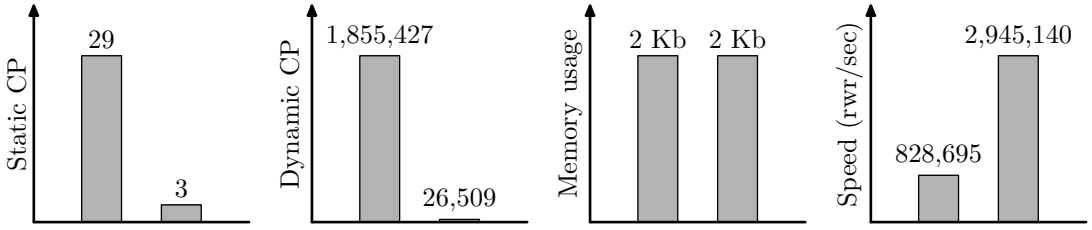
Notons que sans l'optimisation, la complétion d'un programme semble nécessiter un besoin de mémoire proportionnel au nombre de points de choix créés à l'exécution (Dynamic CP). Alors que ce besoin de mémoire devient constant : 1 Kb, lorsque l'analyse du déterminisme est activée.

– minela est un mini-interpréteur ELAN écrit en ELAN. Il permet d'exécuter un programme composé uniquement de règles conditionnelles. Pour un terme clos et un programme donnés, son exécution permet une forme normale et un terme de preuve associé à la dérivation.



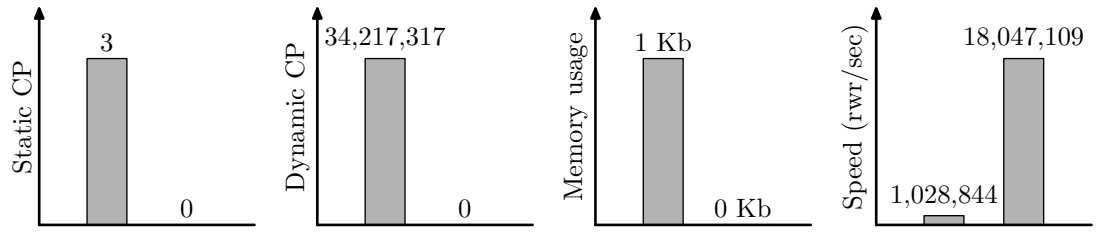
Ici, l'accélération est inférieure à celles des autres exemples, mais c'est principalement dû à la nature de l'application : minela simule de la réécriture non-déterministe et manipule de nombreux termes qui comportent chacun plusieurs milliers de symboles. La proportion de temps passé dans la gestion des points de choix devient ainsi plus petite devant le temps passé à gérer la mémoire par exemple, d'où une accélération moins grande.

– queens est une implantation du problème des n-reines qui cherche une solution pour $n = 14$. C'est également un test classique pour évaluer les performances d'un langage de programmation logique.



Il est intéressant de constater que la diminution du nombre de points de choix créés dynamiquement est proportionnelle à la diminution du nombre de points de choix générés statiquement, et que la vitesse d'exécution s'en trouve inversement améliorée. L'utilisation mémoire reste constante parce qu'elle correspond, dans les deux cas, aux 14 générateurs utilisés pour énumérer les différentes configurations.

– fib est un programme, dans un style purement fonctionnel, qui calcul le 33$^{\text{e}}$ nombre de Fibonacci. Une fois encore, c'est un test typique pour évaluer les performances des langages fonctionnels.



Lorsque l'analyse du déterminisme est activée, le programme généré ne contient plus aucune instruction setChoicePoint, ce qui élimine tout risque de retour arrière. C'est pourquoi la mémoire nécessaire pour sauver les environnements est réduite à 0 Kb. On peut remarquer que la vitesse d'exécution s'en trouve améliorée : plus de 18 millions de règles appliquées par seconde.

Ces résultats expérimentaux montrent clairement que l'analyse du déterminisme permet de réduire le nombre de points de choix posés à l'exécution tout en améliorant la vitesse générale des programmes générés. Mais l'analyse du déterminisme permet aussi de réduire considérablement la mémoire nécessaire pour mémoriser les environnements. C'est d'autant plus important qu'en pratique, cette mémoire a une taille fixe (1000 Kb par exemple) et qu'un manque de mémoire provoque l'arrêt immédiat du programme (la pile utilisée pour sauver les environnements ne peut pas être agrandie pendant l'exécution). Lorsqu'on active l'analyse du déterminisme, la taille mémoire nécessaire est souvent ramenée à une constante indépendante du terme à réduire. Ce qui permet généralement d'exécuter les programmes qui provoquaient un dépassement de mémoire lorsque l'optimisation n'était pas activée.

Troisième partie

Implantation d'un compilateur

Chapitre 10

Architecture logicielle

10.1 Compilation modulaire et compilation séparée 143
10.2 Organisation du compilateur 147
10.3 Fonctionnement du compilateur 150

Une des particularités de la compilation est d'amener les informaticiens à travailler en permanence en présence de plusieurs paradigmes de programmation. Il faut d'une part étudier les schémas de traduction qui vont permettre de transformer les constructions du *langage source* en des constructions du *langage cible*, mais il faut aussi s'intéresser à la façon d'exprimer ces schémas de traduction dans un troisième formalisme : le *langage d'implantation*, qui n'est pas nécessairement relié aux deux premiers. La réalisation d'un compilateur ne se limite pas à l'étude des techniques de traduction, il faut aussi s'assurer que les schémas de traduction proposés et la structure du code généré sont bien en accord avec les normes « implicites » du langage cible. On sait par exemple que le langage `C` permet d'écrire des programmes contenant plusieurs centaines de milliers de lignes, mais il faut aussi savoir que les outils de compilation mettent en œuvre des phases d'optimisation locales et globales qui ont une complexité polynomiale en temps et en espace par rapport à la taille des fonctions à compiler. Il est donc préférable de définir plusieurs fonctions comportant un nombre « raisonnable » de lignes plutôt qu'une seule fonction qui comporterait un « très grand » nombre de lignes. La limite entre « raisonnable » et « très grand » n'est pas rigoureusement définie, puisqu'elle évolue en fonction de la puissance des calculateurs et des algorithmes utilisés par les compilateurs. La pratique montre que la compilation de plusieurs centaines ou milliers de lignes ne pose aucun problème, alors qu'il n'est pas toujours possible de compiler une fonction comportant plusieurs dizaines de milliers de lignes. Dans ce cas, il faut décomposer la fonction `C` en sous-fonctions qui peuvent même être définies dans des fichiers différents.

Dans ce chapitre nous étudions les difficultés de compilation modulaire liées aux langages de programmation par réécriture. Nous présentons ensuite la hiérarchie des classes Java qui composent le compilateur et nous rappelons l'ordre dans lequel les différentes étapes de compilation sont effectuées.

10.1 Compilation modulaire et compilation séparée

Les problèmes liés à la compilation modulaire sont nombreux (Crelier 1994) et amènent à se poser les questions suivantes : comment analyser séparément les modules du langage source pour

construire leur représentation intermédiaire dans un format tel que l'Efix ? Comment compiler séparément chaque module codé dans sa représentation intermédiaire ? Comment éviter, à chaque cycle de compilation, de recompiler les modules n'ayant pas été modifiés ?

Analyse module par module

Comme mentionné dans le chapitre 2, l'analyse syntaxique des langages tels qu'ELAN est rendue difficile par la présence d'opérateurs dont la syntaxe est définie par des règles de grammaire hors contexte. Pour analyser les règles de réécriture d'un module donné, il faut avoir connaissance des règles de grammaire associées à l'ensemble des opérateurs utilisés dans le module, et il faut aussi être capable de construire dynamiquement un parseur dépendant de ces règles de grammaire.

L'aspect modulaire d'ELAN fait que la syntaxe de certains opérateurs peut être définie dans un module autre que celui considéré. Le parseur doit donc parcourir la clôture transitive des importations du module courant pour construire incrémentalement un parseur capable d'analyser les règles de réécriture contenues dans le module.

La complexité des algorithmes et des techniques à mettre en œuvre est telle, qu'à ce jour, seul ASF+SDF (Deursen et al. 1996, Visser 1997) possède un parseur capable d'analyser séparément chaque module pour construire leur représentation intermédiaire asFix. Les méthodes présentées dans (Visser 1997) sont suffisamment générales pour pouvoir être réutilisées et adaptées aux langages Maude, CafeOBJ ou ELAN par exemple. Mais remarquons qu'il n'est pas nécessaire d'utiliser un analyseur modulaire pour pouvoir analyser complètement une spécification et générer les modules associés en représentation intermédiaire : il suffit de lire l'ensemble des règles hors contexte définies dans les différents modules et de construire dynamiquement un parseur (avec l'algorithme d'Earley par exemple), pour pouvoir analyser chaque module séparément. C'est la solution qui a été retenue pour ELAN.

Découpage en modules

La version actuelle d'ELAN permet de lire une spécification et de construire sa représentation au format REF. Mais le caractère monolithique de ce format donne un aspect figé à notre environnement, ce qui rend difficile l'échange de termes et les manipulations de modules telles que leur compilation, leur affichage ou leur transformation par évaluation partielle par exemple. C'est pourquoi nous avons étudié une représentation plus modulaire : l'Efix, présentée dans le chapitre 3. L'Efix est un format qui correspond à la syntaxe abstraite du langage et qui permet de représenter n'importe quelle construction du langage : une relation d'importation, une règle de grammaire hors contexte, un ensemble de règles de réécriture, une stratégie ou un module tout entier, par exemple. Mais devant une telle souplesse, se pose la question de savoir quelle granularité et quel découpage adopter pour représenter un programme.

Doit-on représenter l'ensemble du programme par un seul terme Efix ? Doit-il y avoir une bijection entre les modules ELAN et les modules Efix ? Ou doit-on adopter une structure complètement différente consistant à représenter les règles de grammaire hors contexte, les règles de réécriture et les stratégies par des termes Efix différents ?

Il n'y a vraisemblablement pas de réponse universelle, simplement parce que chaque choix a des avantages qui dépendent des traitements à effectuer par la suite. Pour afficher ou modifier la structure d'un module, par exemple, il est préférable de faire correspondre un terme Efix à chaque module ELAN, mais pour compiler un programme, la notion de module n'est plus nécessaire et il est préférable de regrouper les fonctions commençant par un même symbole de tête dans un

unique terme Efix.

Notons que le format Efix est suffisamment riche pour permettre de passer d'une représentation à l'autre, le choix n'est donc pas primordial. Il est alors naturel de choisir la deuxième représentation mettant en bijection chaque module ELAN avec un module Efix. Cette solution est la plus facile à mettre en œuvre et elle a l'avantage de conserver la structure du programme originel. C'est d'ailleurs la solution qui a été retenue dans le projet ASF+SDF, par exemple.

Génération modulaire

À la différence de nombreux langages de programmation impérative, logique ou fonctionnelle, les langages de programmation par réécriture permettent de « répartir » la définition d'une fonction dans plusieurs modules. Il est en effet possible de définir, dans deux modules différents, des règles dont les membres gauches commencent par un même symbole. Et c'est cette souplesse qui rend difficile, voire impossible, la compilation modulaire, c'est à dire compiler un module sans accéder aux autres modules. Comme nous l'avons vu dans les chapitres 5, 6 et 8, il faut pouvoir regrouper les définitions des règles dont les membres gauches commencent par des symboles identiques pour pouvoir utiliser des algorithmes de filtrage *many-to-one* et offrir une méthode de normalisation efficace.

Nous sommes en présence de deux catégories de modules :

- les *modules syntaxiques*, qui correspondent à des fichiers physiques, et dont le découpage se fait en fonction des sortes et de la syntaxe des opérateurs définie sur ces sortes ;
- les *modules sémantiques*, dont le découpage est relié à la notion de système de réécriture. Ces modules correspondent à des ensembles de règles non nommées dont le membre gauche commence par un même symbole, et à des ensembles de règles nommées apparaissant dans une même stratégie.

La première catégorie est liée à la syntaxe du programme, alors que la deuxième est reliée à sa sémantique. Et il n'y a rien d'étonnant à voir ces deux oppositions apparaître lorsqu'on essaie de traduire des modules du langage source (appartenant à la première catégorie) en des modules du langage cible (appartenant à la deuxième catégorie).

Afin de rendre possible la compilation modulaire, tout en explicitant le passage d'une catégorie à l'autre, il est nécessaire d'introduire une étape de réorganisation (souvent appelée *reshuffling*). Partant d'un ensemble de modules Efix correspondant à des modules ELAN, l'étape de réorganisation analyse ces modules et génère un nouvel ensemble de modules Efix où les règles et stratégies sont regroupées en unités sémantiques. Cette étape garantit, par exemple, que toutes les règles dont les membres gauches commencent par un même symbole, sont regroupées dans un même module. On imagine alors comment appliquer les méthodes proposées dans le chapitre 8, pour compiler séparément chaque module. Chaque ensemble de règles et chaque stratégie apparaissant dans un même module peuvent être compilés en une fonction du langage cible (voir figure 10.1).

Il reste cependant à résoudre un problème apparaissant au cours de la construction du terme réduit : le terme est construit récursivement et suivant le type de symbole à construire (symbole constructeur ou défini), de la mémoire est allouée ou des appels de fonctions sont effectués. Mais comment savoir si un symbole est constructeur ou non, et comment connaître le nom des fonctions à appeler ?

Dans un cadre extrême de compilation modulaire, il n'est possible de savoir qu'un symbole est constructeur que dans le module où le symbole est défini. Pour compiler un module donné, une solution consiste à considérer que tous les symboles non locaux sont des symboles définis (rappelons qu'un symbole défini est un symbole apparaissant en tête d'un membre gauche de

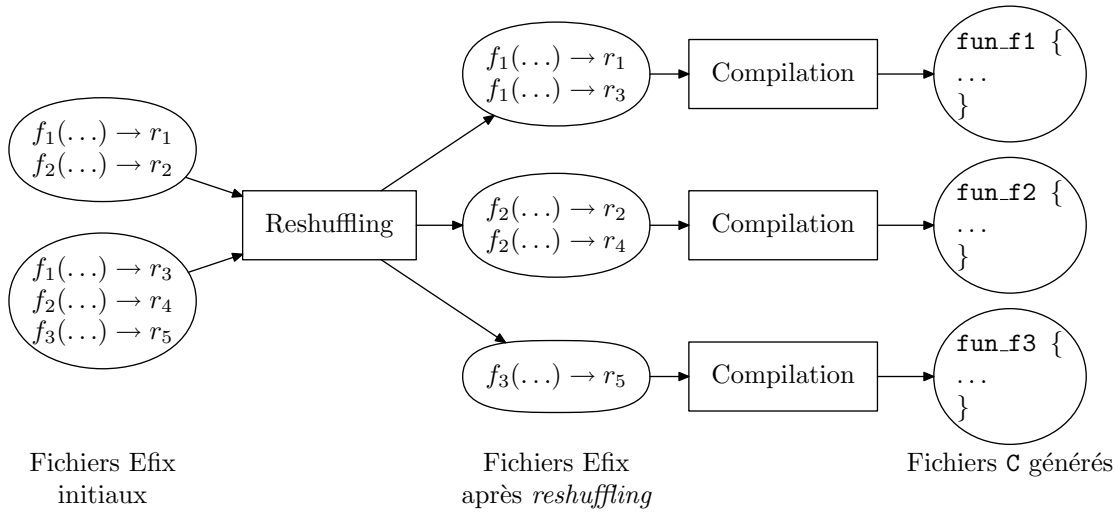


FIG. 10.1 – *Cette figure illustre un schéma de compilation modulaire qui consiste dans un premier temps à réorganiser les modules initiaux pour regrouper les règles commençant par un même symbole. Les nouveaux modules obtenus sont ensuite compilés pour engendrer la création de modules C.*

règle) et qu'ils se construisent en appelant une fonction. Et lorsqu'un symbole local est un constructeur, il faut générer une fonction permettant de construire effectivement le symbole. Cette approche permet de compiler séparément les modules, mais l'efficacité du programme généré est relativement mauvaise dans la mesure où de nombreuses optimisations ne peuvent plus être effectuées. Les constantes ne peuvent plus être partagées, par exemple, ce qui augmente considérablement le nombre d'allocations dynamiques de mémoire.

Les solutions retenues en pratique, pour savoir si un symbole est constructeur, sont moins extrêmes et consistent à gérer une table *globale* qui indique, pour tout symbole, s'il est constructeur ou non. Lorsqu'on ajoute une règle par exemple, un symbole qui était constructeur peut devenir défini (et inversement lorsqu'on supprime une règle), il faut alors recompiler tous les modules qui utilisent ce symbole. Ce compromis, entre vitesse de compilation et vitesse d'exécution, semble être le prix à payer pour permettre de générer un code relativement efficace.

Quant au deuxième problème, qui consiste à savoir comment appeler une fonction définie dans un module externe, il existe principalement deux solutions :

- si le langage cible permet d'utiliser des noms longs pour nommer les fonctions, pour un ensemble de règles définies dans un module, la fonction associée peut prendre pour nom, le nom du module suivi de la signature du symbole de tête des membres gauches des règles ;
- si les noms des fonctions du langage cible ont une taille maximale, il faut alors utiliser un mécanisme d'indirection. Une table, initialisée au lancement du programme, permet d'associer une fonction à chaque nom long (nom du module suivi de la signature du symbole). Ce mécanisme d'indirection est alors utilisé à chaque appel de fonction, ce qui diminue évidemment la vitesse d'exécution du programme généré.

Compilation séparée

Lorsque tous les modules Efix sont traduits en des modules du langage cible, il reste à les compiler séparément en utilisant un compilateur du langage cible.

Pour profiter pleinement des possibilités offertes par les compilateurs C, il faut veiller à ce que le code C généré, d'une compilation à l'autre, soit relativement stable. L'idéal serait qu'une modification effectuée dans un module source entraîne la compilation de ce seul module et que seul le code C associé ait besoin d'être recompilé. Mais on a vu que l'utilisation d'une table globale, pour identifier les constructeurs, pouvait dans certains cas, entraîner la recompilation de tous les modules utilisant un symbole changeant de catégorie. Dans la pratique, cette situation se produit assez rarement, et généralement, lorsqu'un module est modifié, c'est souvent pour corriger la définition d'une règle ou pour ajouter une nouvelle règle.

Après chaque modification d'un module, l'étape de réorganisation (*reshuffling*) doit être appliquée, ce qui produit un nouvel ensemble de modules. Mais on s'aperçoit qu'il est inutile de recompiler la totalité de ces modules parce qu'ils correspondent, dans la grande majorité des cas, aux modules générés par l'étape précédente. Il suffit alors de ne recompiler que les modules qui sont différents : c'est précisément ceux qui contiennent les règles ajoutées ou modifiées. La recompilation de ces modules engendre de nouveaux fichiers C qui sont à leur tour recompilés.

La compilation modulaire permet ainsi de réduire le nombre de compilations et le temps d'attente d'un cycle de compilation à l'autre. Il faut cependant noter que la compilation modulaire n'a pas que des avantages. Comme on l'a vu précédemment, elle empêche d'avoir une vue globale du programme à compiler, ce qui limite les possibilités d'optimisation, telles que l'analyse du déterminisme par exemple. En effet, comment connaître le d-mode d'une stratégie lorsque celle-ci est définie dans un module différent ?

Contrairement à ASF+SDF qui produit de l'asFix, la version actuelle d'ELAN ne permet pas encore d'engendrer un format modulaire tel que l'Efix, et seul le format REF est disponible. C'est pour ces différentes raisons que le compilateur n'est pas encore modulaire. Nous avons cependant adopté une approche hybride qui permet de réduire le temps de compilation tout en ayant une vue globale du programme à compiler. À chaque étape de compilation, l'ensemble du programme REF est engendré et compilé, mais le code C généré est relativement stable d'une compilation à l'autre. Et lorsqu'une règle non nommée est modifiée, par exemple, seul le module C correspondant à la définition de la règle a besoin d'être compilé.

L'approche hybride d'ELAN ne fait pas figure d'exception. Il est en effet connu, lorsqu'on réalise un compilateur qui engendre du langage C, que le temps de compilation du langage source vers le langage C est souvent « court » comparé au temps nécessaire pour compiler les fichiers C générés. C'est pourquoi, de nombreux compilateurs ont une approche similaire, qui consiste à avoir une première phase de compilation globale pour permettre un grand nombre d'optimisations, et une seconde phase de compilation séparée pour réduire le temps de compilation. On peut par exemple citer le compilateur GNU Eiffel (Colnet, Coucaud et Zendra 1998, Zendra, Colnet et Coucaud 1998) qui est sûrement un des compilateurs Eiffel les plus rapides et qui génère des exécutables d'une grande qualité, grâce à son approche hybride lui permettant d'effectuer une optimisation globale.

10.2 Organisation du compilateur

L'implantation actuelle du compilateur est écrite en Java et se décompose en plusieurs classes. Chaque classe correspond à un concept. Parmis ces concepts, certains sont moins généraux que d'autres et peuvent même se voir comme des spécialisations d'un concept plus général. Au niveau de l'implantation, cela se traduit par une notion d'héritage entre les classes. La hiérarchie suivante

présente les classes principales ainsi qu'un ou deux niveaux du graphe d'héritage :

- `REFParser` : c'est le parseur qui permet de lire et d'analyser une spécification au format REF. Cet analyseur a été réalisé à l'aide du générateur de parseurs JavaCC.
- `REM` (`Reduce Elan Machine`) : c'est la classe qui coordonne les opérations à effectuer (lire la spécification au format REF, compiler les règles et les stratégies, puis générer le programme `C`).
- `RewriteRule` : les instances de cette classe sont des règles de réécriture, qui sont représentées par un membre gauche (de la classe `Term`), un membre droit, et une liste d'évaluations locales (de la classe `BranchEvaluation`).
- `Term` : cette classe permet de représenter des termes et définit de nombreuses opérations telles que le comptage des variables, le renommage ou l'aplatissement. Dans le compilateur, un terme est représenté par un symbole (de la classe `Symbol`), un tableau de sous-termes et une multiplicité.
- `BranchEvaluation` : cette classe permet de représenter un ensemble d'évaluations locales apparaissant dans une règle de réécriture ou dans une branche `try` de la construction `choose try ... end`. Elle est constituée d'un tableau d'évaluations locales (de la classe `LocalEvaluation`).
- `LocalEvaluation` : c'est une interface qui définit la notion d'évaluation locale. Dans l'implantation courante, trois classes permettent de définir les trois types d'évaluations locales définis dans le langage :
  - `Condition` permet de représenter les conditions de la forme **if** c par un terme (de la classe `Term`) ;
  - `LocalAffectation` permet de représenter les conditions de filtrage de la forme **where** $p := (S)c$;
  - `Choice` représente les alternatives (`choose`) de la construction `choose try ... end`. Il s'agit d'un tableau d'ensembles d'évaluations locales (de la classe `BranchEvaluation`).
- `StrategyTerm` : c'est une classe abstraite permettant de représenter des expressions construites à partir d'opérateurs élémentaires de stratégies :
  - `StrategyChoose` est un opérateur de choix pouvant s'appliquer sur des règles ou des stratégies :
    - `StrategyOneRule` correspond au dc one$(r_1, \ldots, r_n)$;
    - `StrategyDcRule` correspond au dc$(r_1, \ldots, r_n)$;
    - `StrategyDkRule` correspond au dk$(r_1, \ldots, r_n)$;
    - `StrategyOneStrat` correspond au dc one$(S_1, \ldots, S_n)$;
    - `StrategyDcStrat` correspond au dc$(S_1, \ldots, S_n)$;
    - `StrategyDkStrat` correspond au dk$(S_1, \ldots, S_n)$.
  - `StrategyCons` correspond à la concaténation « ; » ;
  - `StrategyRepeat` correspond au constructeur repeat* ;
  - `StrategyIterate` correspond au constructeur iterate* ;
  - `StrategyFail` correspond à la stratégie fail ;
  - `StrategyId` correspond à la stratégie id ;
  - `StrategyEval` correspond à un méta-interpréteur de stratégies décrit dans la thèse de Peter Borovanský (1998) ;

  - `StrategyMeta` correspond à une version restreinte de la stratégie meta-apply présentée dans le chapitre 1 et décrite dans (Borovanský 1998) ;

- `Flatterm`, `DDNode`, `DDTree` et `ACDDTree` sont quatre classes qui permettent de représenter les arbres de filtrage syntaxique et les structures de filtrage AC :
  - `Flatterm` permet de représenter un *terme* vu comme une suite de symboles (voir paragraphe 5.1). Son implantation se compose d'un symbole (de la classe `Symbol`) et de liens (de la classe `Flatterm`) vers le symbole précédent, suivant et la fin de portée du symbole courant. Cette structure de termes est présentée en détail dans (Christian 1993).
  - `DDNode` représente un *nœud* d'un arbre de filtrage (ou un état d'un automate de filtrage). Il se compose d'un tableau de nœuds (de la classe `DDNode`) pour représenter les différentes règles de transition d'états.
  - `DDTree` correspond à un arbre de filtrage.
  - `ACDDTree` correspond à une structure de filtrage AC.

- `Symbol` : cette classe abstraite permet de représenter les différents types de symboles dans la représentation abstraite d'un programme REF. On peut distinguer deux catégories de symboles :
  - `SymbolCode` est une classe qui permet de représenter les symboles pour lesquels il existe une règle de grammaire hors contexte définissant leur signature. Dans une spécification, le nombre de ces symboles est toujours fini, ce qui nous permet de leur associer un numéro unique, appelé *code*, d'où le nom `SymbolCode`. On peut distinguer quatre sous-catégories de symboles :
    - `SymbolAC` qui représente les symboles AC ;
    - `SymbolFree` qui représente les symboles de la théorie vide ;
    - `SymbolVariable` qui représente les variables ;
    - `SymbolBuiltin` qui représente les symboles dont la sémantique est prédéfinie par le langage ELAN. Les constantes $\top$ et $\bot$ (`true` et `false`) par exemple, ont une syntaxe libre (définie dans un module ELAN), mais leur sémantique est imposée et doit correspondre aux valeurs booléennes de vérité.
  - `SymbolValue` est une classe qui permet de représenter les symboles dont la syntaxe et la sémantique sont définies par le langage. Ces symboles font toujours partie d'un ensemble infini de symboles et c'est pour cela que la syntaxe ne peut pas être décrite à partir d'un nombre fini de règles de grammaires hors contexte. Dans le langage ELAN, on distingue trois catégories de tels symboles :
    - `SymbolInteger` permet de représenter les entiers. `1`, `2` et `3` sont des entiers de la classe `SymbolInteger` par exemple ;
    - `SymbolIdentifier` permet de représenter les identificateurs. Dans le langage, les identificateurs correspondent aux suites de caractères alphanumériques. `a`, `b` et `plus` sont des identificateurs de la classe `SymbolIdentifier` par exemple ;
    - `SymbolString` permet de représenter les chaînes de caractères. Ce sont des suites quelconques de caractères qui commencent et se terminent par des guillemets. `"hello"` et `"le résultat est 3"` sont des chaînes de caractères de la classe `SymbolString` par exemple.

- `Lexem` : cette classe abstraite permet de représenter les unités lexicales qui sont utilisées pour construire la représentation abstraite d'un programme REF. On peut distinguer différentes sous-classes de lexèmes :
  - `LexemChar` permet de représenter un caractère ;
  - `LexemIdentifier` permet de représenter un identificateur ;
  - `LexemModule` permet de représenter un nom de module ;
  - `LexemNum` permet de représenter un entier ;
  - `LexemRuleName` permet de représenter un nom de règle ;
  - `LexemSort` permet de représenter un nom de sorte ;
  - `LexemStrategyName` permet de représenter un nom de stratégie ;
  - `LexemVariableName` permet de représenter un nom de variable.

Dans sa version courante, le compilateur de REF se compose de 80 classes Java, ce qui représente 15.000 lignes de code environ.

10.3 Fonctionnement du compilateur

La compilation d'un programme REF se décompose en trois grandes phases : la lecture et la représentation interne d'un programme REF, l'application de pré-traitements sur cette représentation abstraite du programme, et la compilation des règles et des stratégies.

Lecture et représentation interne d'un programme REF

Cette première étape est essentiellement réalisée par le parseur implanté par la classe `REFParser`. Elle consiste principalement à lire un programme REF et à construire une collection d'objets qui représentent les constructions reconnues.

Considérons, par exemple, le morceau de texte suivant :

```
RULE(
⌈extractrule1⌉,⌈Element⌉,⌈liste⌉,
FSYM(FSYM(VAR(0,⌈Element⌉).VAR(1,⌈Liste⌉).nil, ⟨@.@⟩).nil, ⟨extract(@)⟩,
VAR(0,⌈Element⌉),
nil)
```

Cette expression REF a été présentée au paragraphe 3.1, page 42, et elle correspond au codage de la règle de réécriture nommée :

```
[extractrule1] extract(element.liste) => element        end
```

Le parseur lit une suite de lexèmes `RULE`, `(`, ⌈`extractrule1`⌉, ... , `FSYM`, `VAR`, `.`, `nil`, *etc.* et reconnaît qu'il s'agit de la définition d'une règle de réécriture nommée, composée de deux termes (le membre gauche et le membre droit) et d'une liste vide d'évaluations locales.

Les termes ⟨`extract(@)`⟩`(VAR(0)`⟨`@.@`⟩`VAR(1))` et `VAR(0)`, qui correspondent, à un renommage près, aux termes `extract(element.liste)` et `element`, sont alors construits et la règle toute entière peut être représentée en mémoire. Il faut pour cela créer les unités lexicales et les symboles qui apparaissent dans les termes et la règle. On peut par exemple citer ⌈`extractrule1`⌉ qui est un lexème de classe `LexemRuleName`, ⟨`extract(@)`⟩ qui est un symbole de la classe `SymbolFree` (composé de quatre lexèmes appartenant aux classes `LexemIdentifier` et `LexemChar`) et le symbole `VAR(0)`, de la classe `SymbolVariable`, qui est composé d'un lexème de la classe `LexemVariableName`.

En même temps que la représentation abstraite d'un programme REF se construit, certaines transformations sont effectuées en parallèle afin de simplifier les traitements ultérieurs. Les règles sont, par exemple, regroupées en fonction de leur nom ou du symbole de tête du membre gauche, dans le cas d'une règle non nommée. Les termes comportant des symboles AC sont ensuite aplatis et mis en forme canonique, et les règles dont le membre gauche est non-linéaire sont transformées en des règles linéraires conditionnelles (les variables qui apparaissent plusieurs fois dans le membre gauche sont renommées et leur égalité est testée par des conditions). Ces étapes de transformation permettent de détecter les règles qui n'appartiennent pas aux classes de motifs définis dans le chapitre 6 et pour lesquelles un algorithme de filtrage AC plus général doit être utilisé, par exemple.

Lorsque l'intégralité du programme REF est lue et que sa représentation abstraite est construite en mémoire, on peut considérer que toutes les constructions la composant peuvent être compilées et traduites dans le langage cible.

Décoration de la représentation abstraite d'un programme REF

Au cours de cette deuxième étape, les règles et les stratégies sont analysées pour préparer et simplifier l'étape de génération de code. C'est à ce moment-là, par exemple, que les automates de filtrage sont construits et associés aux ensembles de règles correspondants. C'est aussi au cours de cette étape que l'analyse du déterminisme est effectuée pour associer à chaque règle et à chaque stratégie son d-mode.

Une des difficultés de cette phase intermédiaire est d'associer un nom de variable (du langage cible) à chaque symbole apparaissant dans une règle. Ceci afin de mémoriser une substitution résultant de l'étape de filtrage, ou un résultat intermédiaire obtenu au cours de la construction du terme réduit par exemple. Lorsqu'on donne ces noms, il faut veiller à minimiser le nombre de noms utilisés pour réduire la taille des environnements sauvegardés par les primitives de gestion du non-déterminisme, et il faut aussi veiller à ce qu'un même nom de variable soit associé aux sous-termes qui apparaissent plusieurs fois dans une expression. Ce problème de nommage est à comparer avec les problèmes d'allocations de registres (Aho et al. 1989, Wilhelm et Maurer 1994) dans la compilation des langages impératifs par exemple.

Compilation des règles et des stratégies

Cette dernière phase intervient après la construction des règles en mémoire, l'aplatissement des termes contenant des symboles AC, l'analyse du déterminisme, la construction des automates et des structures de filtrage AC, et après la détection des sous-termes partagés et l'allocation des noms de variables. Il ne reste alors qu'à générer du code cible correct.

Les ensembles de règles non nommées et les stratégies sont successivement compilés en des fonctions `C`. Chacune de ces fonctions suit le schéma de compilation proposé dans le chapitre 8 : elle comporte une phase de filtrage, une phase de sélection de règles ou de stratégies, une phase de calcul des évaluations locales et une phase de construction du terme réduit. Chaque étape de compilation est indépendante, les fonctions générées peuvent s'écrire dans des fichiers différents, on peut ainsi noter que la compilation des ensembles de règles et des stratégies peut se faire en parallèle lorsqu'on dispose d'une machine multi-processeurs par exemple.

Chapitre 11

Support d'exécution

11.1 Structures de données 153
11.2 Opérations internes 155
11.3 Sortes et opérations prédéfinies 156
11.4 Gestion de la mémoire 158
11.5 Synthèse 163

Lorsqu'on réalise un compilateur, il y a des constructions du langage source qui peuvent se compiler en des fonctions identiques, quel que soit le contexte d'utilisation. Il faut alors choisir entre générer systématiquement du code cible ou bien intégrer la fonction dans une bibliothèque de support d'exécution et l'appeler lorsque c'est nécessaire. Considérons la fonction `printf` du langage `C`, par exemple. Cette fonction n'est pas compilée à chaque fois qu'une instruction d'affichage est nécessaire, mais elle fait partie d'une bibliothèque (`libc`) qui est couplée avec chaque programme généré par le compilateur `C`.

Dans le cadre du compilateur ELAN, nous avons défini une bibliothèque qui regroupe non seulement des opérations utilisées par le code généré, telles que des fonctions d'affichage, de construction de termes ou de résolution d'un graphe biparti par exemple, mais nous avons aussi prédéfini un certain nombre de types de données tels que les termes, les graphes bipartis compacts ou les vecteurs de bits. Ce chapitre présente les structures de données et les fonctions principales de cette bibliothèque, ainsi que les choix d'implantation qui ont été faits.

11.1 Structures de données

Représentation des termes. Les termes du premier ordre sont des objets fondamentaux des langages de programmation algébriques. Dans les implantations, leur représentation est souvent dérivée d'une structure arborescente. Il existe cependant une alternative, proposée par Jim Christian (1993), qui consiste à utiliser une structure linéaire. Ces termes, appelés *flatterms*, sont alors représentés par une liste simplement ou doublement chaînée. Cette structure permet d'améliorer l'efficacité des procédures de parcours, par rapport aux représentations arborescentes classiques. L'utilisation de *flatterms* entraîne cependant des restrictions qui rendent leur utilisation impossible dans le cadre de notre compilateur : il est difficile de représenter les symboles AC, qui ont une arité variable, et le partage de sous-termes est impossible. C'est pourquoi nous avons étudié différentes solutions fondées sur des structures d'arbre, et nous avons choisi une représentation permettant d'utiliser des symboles AC d'arité variable et n'introduisant pas de surcharge

lorsqu'aucun symbole AC n'est utilisé. Les sous-termes de symboles syntaxiques sont mémorisés dans un tableau de taille fixe alors que les sous-termes de symboles AC sont mémorisés dans une liste simplement chaînée (voir figure 11.1), ce qui facilite les opérations d'insertion et de fusion de listes triées.

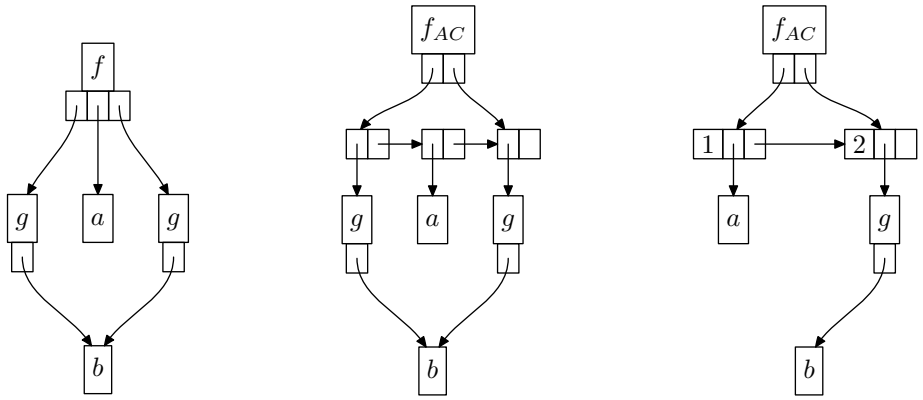


FIG. 11.1 – *Considérons les termes $f(g(b),a,g(b))$ et $f_{AC}(g(b),a,g(b))$, où f_{AC} est un symbole AC. Leur représentation est illustrée par les deux premiers dessins de la figure. La structure de données peut facilement être étendue pour représenter des termes en forme canonique : le troisième dessin montre comment les occurrences multiples d'un même terme sont représentées, en mémorisant la multiplicité dans la liste de cellules.*

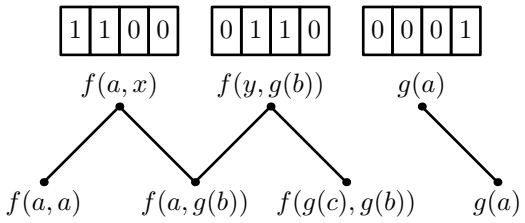
Représentation des vecteurs de bits. Les vecteurs de bits sont des objets largement utilisés dans le cadre de la compilation de la réécriture. Ils servent principalement, pendant le filtrage, à mémoriser les motifs qui filtrent un terme donné. Un indice i est associé à chaque membre gauche de règle, et pour un problème de filtrage donné, le $i^{\text{ième}}$ bit du vecteur est mis à 1 lorsque le motif numéro i filtre le sujet.

L'opération de filtrage étant très fréquemment appliquée, il est essentiel que l'implantation des vecteurs de bits soit la plus optimale possible. Nous avons distingué deux cas, suivant que la taille du vecteur est plus petite ou plus grande que 32, ceci parce que 32 est le nombre de bits utilisé par une grande majorité des processeurs actuels, pour représenter les entiers. Il existe des processeurs dits 64 bits, mais ils sont évidemment capables de manipuler des entiers stockés sur 32 bits.

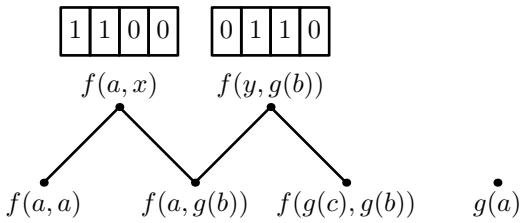
Ainsi, lorsque la taille du vecteur de bits est plus petite que 32, celui-ci est représenté par un entier de 32 bits. Et des fonctions sont définies pour modifier ou tester la valeur d'un bit donné. Lorsque le vecteur a une taille supérieure, il n'est plus possible d'utiliser un entier pour le mémoriser, et c'est pourquoi nous utilisons un tableau d'entiers. L'accès aux différents bits est naturellement plus lent puisqu'il faut accéder auparavant à la bonne case du tableau.

Ces précisions peuvent paraître techniques, mais elles se justifient par l'importance d'une telle représentation. Ces vecteurs de bits sont d'une part utilisés par les procédures de filtrage, mais ils sont aussi utilisés pour représenter les graphes bipartis nécessaires au filtrage AC. Et le fait de pouvoir les représenter par des entiers permet de réduire le nombre d'allocations mémoire et d'accroître les performances globales d'environ 15% par rapport à une implantation qui n'utiliserait que la version généralisée des vecteurs de bits.

Représentation des graphes bipartis compacts. Dans le chapitre 6, nous signalions qu'une structure particulière de graphes bipartis compacts permettait de rendre l'opération d'extraction d'un graphe biparti extrêmement performante. Nous utilisons effectivement un tableau de vecteurs de bits pour représenter un graphe biparti compact : à chaque sous-motif est associé un vecteur qui indique quels sont les sous-termes du sujet qui sont filtrés par ce sous-motif. Considérons une nouvelle fois les motifs $f_{AC}(z,f(a,x),g(a))$, $f_{AC}(f(a,x),f(y,g(b)))$ et le terme clos $f_{AC}(f(a,a),f(a,g(b))$, $f(g(c),g(b))$, $g(a))$. Le graphe biparti compact est représenté par trois vecteurs de bits :



Avec cette représentation, l'extraction d'un graphe biparti se fait en sélectionnant un sous-ensemble des vecteurs de bits. La sélection des deux premiers vecteurs, par exemple, permet de construire le graphe biparti associé au deuxième motif $f_{AC}(f(a,x),f(y,g(b)))$:



11.2 Opérations internes

Le code généré par le compilateur suppose qu'un certain nombre d'opérations sont prédéfinies, parmi lesquelles on peut citer :

- `term_alloc` et `term_add_onf` qui permettent respectivement de construire un symbole constructeur et d'ajouter un sous-terme à un symbole AC pour calculer directement la forme canonique du terme ;
- `normalise` qui permet de re-normaliser un terme construit en mémoire. Cette opération ne devrait pas être nécessaire lorsqu'on suppose que les termes sont construits en appliquant une stratégie *leftmost-innermost*. Il existe cependant dans ELAN, une opération qui permet de remplacer un sous-terme par un autre, et dans ce cas, le terme résultat n'est plus forcement irréductible, d'où la nécessité de re-normaliser le terme ;
- `term_cmp` et `merge_sorted_list` qui permettent de comparer deux termes et de fusionner des listes triées de termes pour calculer une forme canonique par exemple.

Nous ne pouvons pas énumérer ici l'ensemble de fonctions qui composent la bibliothèque dans la mesure où celle-ci comporte environ 10.000 lignes de `C`, mais citons encore deux autres fonctions, `maximal_extract_fail` et `next_pe_extract_fail`, qui sont intéressantes dans la mesure où leur exécution modifie le contrôle de flot du programme généré. Ces deux fonctions permettent d'instancier une variable se trouvant directement sous un symbole AC, en énumérant les partitions de l'ensemble des termes non « capturés » par la résolution des graphes bipartis. La particularité de ces fonctions est qu'elles utilisent setChoicePoint pour poser des points de choix et gérer l'énumération des solutions : lorsqu'une instance est trouvée, un point de choix est posé

et la solution est retournée. Cette solution est ensuite utilisée par le programme généré pour construire un terme, et son exécution se poursuit. Lorsqu'un échec réveille le point de choix posé par l'une des deux fonctions, leur exécution peut se continuer pour calculer et retourner une autre solution. Lorsque toutes les solutions ont été calculées, un fail est naturellement engendré. Cet exemple montre que le mécanisme de gestion du non-déterminisme est uniforme, que ce soit dans le code généré pour évaluer les conditions ou les stratégies par exemple, ou que ce soit dans la bibliothèque qui n'est pas du code généré. Le point important ici, est que le mécanisme de gestion du non-déterminisme est suffisamment clair pour permettre à un programmeur d'écrire des fonctions utilisant les primitives de gestion de points de choix.

11.3 Sortes et opérations prédéfinies

Lorsqu'on réalise un interpréteur ou un compilateur, il est souvent délicat d'implanter les sortes élémentaires (*builtins*) prédéfinies par le langage de spécification, parce que celles-ci doivent s'intégrer complètement avec les sortes définies par l'utilisateur. Dans le cadre d'ELAN, qui manipule essentiellement des termes, il faut intégrer les chaînes de caractères, les identificateurs et les entiers, par exemple. La difficulté est d'offrir une implantation qui soit d'une efficacité comparable à celle offerte par le langage cible. Dans ce paragraphe, nous proposons d'étudier différentes façons d'implanter la sorte élémentaire représentant les entiers par exemple.

Afin d'obtenir l'implantation la plus efficace, une solution naturelle consiste à représenter les *entiers* d'ELAN par les entiers du langage cible (le langage C). Mais étant donné que les termes sont représentés par une structure arborescente faisant intervenir des pointeurs (voir figure 11.2), le mélange de ces deux types de représentation implique que certains symboles ont des sous-termes représentés par des pointeurs, et d'autres sous-termes représentés par des entiers.

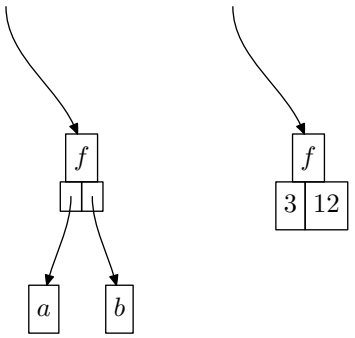


FIG. 11.2 – *Dans la bibliothèque de support d'exécution, les termes sont représentés par une structure arborescente. Le terme $f(a,b)$, par exemple, est ainsi représenté (dessin de gauche) par un pointeur vers une zone de mémoire contenant le symbole f et contenant deux pointeurs vers des zones représentant les constantes a et b. Supposons maintenant que les entiers d'ELAN soient représentés directement par des entiers, le terme $f(3,12)$ serait alors représenté (dessin de droite) par un pointeur vers une zone de mémoire contenant le symbole f et les deux valeurs* 3 *et* 12.

Lorsqu'on définit des fonctions de parcours de termes (dans le cadre du filtrage ou de la comparaison de deux termes par exemple), il est essentiel de pouvoir accéder aux sous-termes.

Mais si la représentation des sous-termes n'est pas toujours la même, l'accès se fait différemment en fonction de leur sorte. Il faut donc connaître la signature d'un symbole pour pouvoir accéder à ses sous-termes. Du point de vue compilation, ce n'est pas gênant, mais du point de vue exécution, c'est un handicap parce qu'il faut accéder en permanence à la signature. De plus, lorsqu'on considère un terme, le seul moyen de connaître sa sorte (terme ou entier ?), est d'accéder à la signature du symbole père, ce qui n'est pas toujours possible. Le problème de cette représentation est que la sorte d'un terme n'est pas codée dans sa représentation.

Il existe cependant une variante d'implantation permettant de distinguer à l'exécution les pointeurs des entiers. Cette variante n'est évidemment pas « pure », parce que liée au fonctionnement des processeurs actuels, mais elle est largement utilisée dans d'autres implantations de langages. L'idée consiste à remarquer que pour des raisons d'alignement, tous les pointeurs sont des multiples de 4 ou de 8, ce qui signifie que les deux derniers bits de leur représentation binaire ont toujours 0 pour valeur. Il suffit alors de représenter les entiers en mettant le dernier bit à 1 (cela s'appelle un *tag*) pour qu'il soit possible de les différencier des pointeurs. Cette approche oblige évidemment à recoder tous les entiers en effectuant un décalage de bits, mais la pratique montre que ces opérations sont efficaces et que l'utilisation d'entiers décorés par un `tag` ne ralentit les opérations que de 10% environ, par rapport à l'utilisation d'entiers « classiques ». C'est cette variante que nous avons choisi d'utiliser dans le cadre d'ELAN, parce qu'elle permet de connaître la sorte d'un terme à l'exécution, ce qui facilite grandement l'écriture des fonctions de la bibliothèque de support d'exécution tout en offrant des performances raisonnables.

Notons quand même que ce type de représentation limite les conditions d'utilisation des entiers. Appelons `builtinInt` la sorte des entiers ainsi codés. Il n'est plus possible de définir des constructeurs de sorte `builtinInt`, sinon le problème se poserait à nouveau : le code généré par le compilateur dépend de la sorte des sous-termes, et pour accéder à un sous-terme de sorte `builtinInt`, le compilateur génère une fonction qui accède directement à l'entier (et non à un pointeur). Si des *termes* de sorte `builtinInt` pouvaient avoir une forme normale qui ne soit pas un entier (au sens $\mathbb{N}$), la fonction d'accès générée ne serait plus correcte puisqu'il faudrait, dans ce cas, accéder aux pointeurs.

Pour lever cette limite, il existe une autre solution qui consiste à « emballer » systématiquement les entiers en utilisant un constructeur interne au compilateur (voir figure 11.3). L'inconvénient d'une telle approche étant de pénaliser considérablement les opérations sur les entiers en introduisant des étapes « de déballage » et « d'emballage ». L'approche choisie dans ELAN est intéressante parce qu'elle offre deux possibilités :

- lorsque la vitesse de calcul est importante, c'est à l'utilisateur de s'assurer que les symboles définis sur les entiers sont bien complètement définis (*i.e.* la forme normale d'un terme de sorte entier doit être un entier, et non un terme), et dans ce cas il peut utiliser le module `builtinInt`. Lorsque cette contrainte n'est pas satisfaite, le code généré pour le filtrage ne peut pas être correct et l'exécution se termine par une erreur ;
- lorsque la sécurité est importante et que l'utilisateur ne veut pas se soucier de la complète définition des symboles utilisés, il peut alors utiliser le module `int` qui utilise le module `builtinInt`, mais introduit un nouveau symbole d'injection (`@ : (builtinInt) int`) permettant « d'emballer » les entiers. Ce qui ressemble à la dernière solution proposée, sauf que le constructeur `@` n'est plus interne au compilateur mais défini en ELAN.

Les opérations élémentaires sur les `int` sont définies en ELAN de la manière suivante :

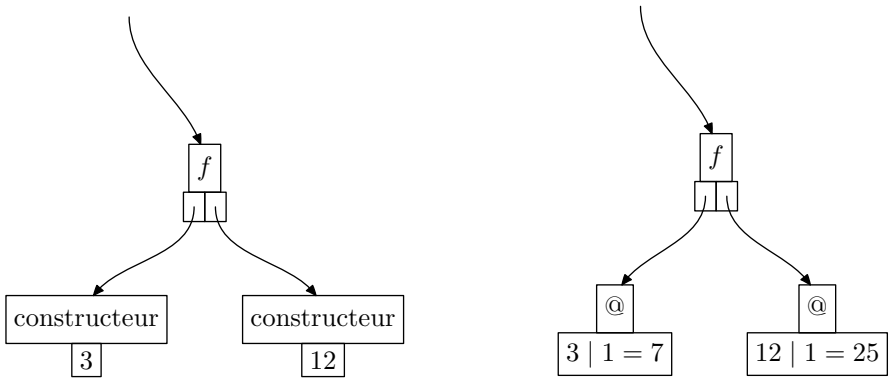


FIG. 11.3 – *Cette figure illustre deux manières « d'emballer » les entiers. Sur le dessin de gauche, le constructeur utilisé est un symbole interne au compilateur qui permet de représenter un entier. Le dessin de droite montre comment un symbole d'injection (`@ : (builtinInt) int`), défini en ELAN, peut-être utilisé pour plonger la sorte `builtinInt` dans la sorte `int` et assurer que tous les objets de sorte `int` sont bien des termes et non des entiers (au sens* $\mathbb{N}$*). Dans le cadre d'ELAN, ce mécanisme s'ajoute à la méthode choisie pour représenter les entiers : la représentation binaire d'un entier* n *est décalée d'un bit vers la gauche et le dernier bit est mis à* 1 *(noté* $n \mid 1$*), ce qui revient à multiplier l'entier* n *par deux et ajouter* 1*.*

```
rules for int
  a,b,c : builtinInt;
global
  [] [a]+[b]   => [c] where c:=()a+b    end
  [] [a]-[b]   => [c] where c:=()a-b    end
  [] [a]*[b]   => [c] where c:=()a*b    end
  [] [a]/[b]   => [c] where c:=()a/b    end
end
```

où `[@]` est un alias de l'opérateur `@ : (builtinInt) int`. Les opérations sur les entiers de sorte `builtinInt` sont quant à elles implantées par des macros du langage `C` :

```
#define fun_plus(a,b)  setIntegerTag(getInt(a) + getInt(b))
#define fun_minus(a,b) setIntegerTag(getInt(a) - getInt(b))
#define fun_mul(a,b)   setIntegerTag(getInt(a) * getInt(b))
#define fun_div(a,b)   setIntegerTag(getInt(a) / getInt(b))
```

où les fonctions `setIntegerTag` et `getInt` servent respectivement à coder et à décoder les représentations binaires des entiers.

11.4 Gestion de la mémoire

Tout comme la gestion du non-déterminisme, la gestion de la mémoire pose elle aussi des problèmes se situant à deux niveaux d'étude distincts. Dans le cadre de la gestion du non-déterminisme, il fallait d'une part étudier la définition de primitives permettant de placer et d'enlever des points de choix, et d'autre part étudier des algorithmes de compilation permettant de minimiser l'utilisation de ces primitives. Dans le cadre de la gestion mémoire, la définition de primitives d'allocation et de restitution de zones de mémoire fait souvent partie intégrante

du langage cible. Le langage `C` propose ainsi deux primitives `malloc` et `free` qui permettent respectivement de réserver un bloc de mémoire et de le rendre au système. Le problème est donc de minimiser l'utilisation de ces primitives et surtout d'éviter les « fuites » de mémoire : tout bloc de mémoire alloué doit être rendu rapidement au système lorsqu'il n'est plus utilisé. Il existe principalement deux approches pour gérer la mémoire d'un système : la gestion explicite des primitives d'allocation et de restitution, ou la mise en place d'un « ramasse miettes » (*garbage collector*) qui possède une vue d'ensemble des zones de mémoire allouées et dont un des rôles est de rendre transparente pour l'utilisateur, la restitution des zones qui ne sont plus utilisées.

Gestion explicite de la mémoire

Pour le programmeur, la gestion explicite de la mémoire revient à déterminer (statiquement) les variables qui référencent des zones de mémoire qui ne sont plus utilisées (dynamiquement). La difficulté de cette analyse dépend des algorithmes à implanter mais aussi des structures de données utilisées. Considérons un modèle relativement simple de programmation, où toutes les fonctions se décomposent en trois étapes :

1. allocation des zones de mémoire nécessaires au calcul ;
2. calcul du résultat ;
3. restitution des zones de mémoire qui ne sont plus utilisées.

Il arrive que tous les objets alloués au début d'une fonction ne soient plus nécessaires une fois le calcul effectué, et dans ce cas, ils peuvent être rendus au système. Mais d'une manière générale, lorsque des zones de mémoire allouées sont utilisées pour construire le résultat de la fonction, ou comme arguments d'autres fonctions, il devient difficile, voire impossible de déterminer statiquement les zones de mémoire qui ne sont plus utilisées. Cela arrive en particulier lorsqu'on manipule des structures de graphes, des structures circulaires ou des structures de termes partagés par exemple.

Considérons une fonction qui simule l'application de la règle $f(x,y) \rightarrow x$ au terme $f(t_1,t_2)$. On imagine alors facilement que cette fonction utilise deux variables (statiques) `x` et `y` qui, à l'exécution, référencent des zones de mémoire représentant les termes t_1 et t_2. Le problème est de savoir si la zone de mémoire contenant t_2 peut être restituée une fois construit le terme réduit référencé par `x`. Il est ici impossible de connaître la réponse, simplement parce que cette zone de mémoire est peut être référencée par des variables utilisées par d'autres fonctions. Pour résoudre ce type de problème, il existe un mécanisme, appelé *compteur de références*, qui consiste à mémoriser dans chaque zone de mémoire allouée, le nombre de variables qui la référencent. Ce nombre est incrémenté ou décrémenté lorsqu'une variable utilise ou n'utilise plus la zone considérée, et quand ce nombre a pour valeur 0, c'est que la zone n'est plus utilisée et peut être rendue au système. C'est un mécanisme qui est habituellement simple à implanter mais relativement peu efficace dans la mesure où un grand nombre de mises à jour de compteurs sont effectuées. De plus, l'utilisation de références ne permet généralement pas de libérer les zones de mémoire occupées par des structures circulaires.

Ce type de gestion mémoire a été expérimenté dans le cadre de la première implantation du compilateur réalisée par Marian Vittek (1996). Mais l'étude expérimentale a montré que les résultats pouvaient être améliorés en termes de sécurité et d'efficacité. Il faut savoir que l'utilisation de compteurs de références ne tolère aucune erreur : si un compteur est incrémenté ou décrémenté par erreur, la gestion mémoire est complètement faussée puisqu'une zone peut ne jamais être rendue au système, ou plus grave encore, une zone utilisée peut être rendue et réallouée pour un autre calcul, ce qui provoque généralement une erreur à l'exécution. Dans certaines situations, ce type de gestion mémoire peut être difficile à stabiliser dans la mesure

où les instructions de gestion mémoire sont réparties dans l'ensemble du programme, ce qui ne facilite pas la recherche d'erreurs. Dans le cadre d'ELAN, la présence de calculs non-déterministes rend encore plus difficile l'utilisation de compteurs de références : lorsqu'un fail est exécuté, cela peut réactiver une fonction qui s'était terminée normalement, mais il faut alors décrémenter les compteurs de tous les termes qui ont été créés depuis la pose du point de choix concerné. On imagine alors la complexité des schémas de compilation mis en œuvre pour gérer le non-déterminisme et la mémoire, et il n'est pas étonnant que ce premier prototype de compilateur ne soit pas parfaitement stable.

D'un point de vue efficacité, la gestion mémoire pouvait elle aussi être améliorée, mais il a fallu attendre une nouvelle implantation du compilateur pour s'en convaincre. Comme mentionné précédemment, pour gérer la mémoire en utilisant des compteurs de références, il faut pouvoir mémoriser un nombre de références dans chaque objet créé. Le caractère particulier de la réécriture fait que les objets alloués, qui sont majoritairement des symboles, sont petits et nombreux (plusieurs milliers d'objets composés de 3 à 4 mots mémoire en moyenne). L'expérience montre que le fait d'ajouter un mot mémoire à chaque objet, pour mémoriser le nombre de références, augmente d'environ 20% la mémoire totale consommée et dégrade d'autant les performances. Cette baisse de performance est principalement due à l'architecture des ordinateurs actuels qui utilise une mémoire cache pour réduire les temps de transfert entre le processeur et la mémoire principale. En effet, l'augmentation générale de la taille des objets manipulés diminue en conséquence le nombre d'objets se trouvant dans la mémoire cache, ce qui augmente les temps de transfert et diminue d'autant les performances globales.

C'est pour remédier aux problèmes d'efficacité et de sûreté que nous avons étudié d'autres approches pour gérer la mémoire.

Utilisation d'un ramasse miettes

Le terme générique de « ramasse miettes » désigne un ensemble de méthodes qui permettent de gérer globalement la mémoire. L'idée consiste à utiliser des heuristiques pour entrelacer des phases de calcul et des phases de récupération de mémoire. Pour cela, l'ensemble des zones de mémoire allouées pendant le calcul sont mémorisées, et en fonction de critères, qui dépendent du temps séparant deux phases de récupération ou de la proportion de mémoire allouée, par rapport à l'espace total disponible, une étape de récupération est déclenchée. Cela consiste à détecter les zones de mémoire qui ne sont plus utilisées et à les rendre au système.

Plusieurs algorithmes de ramasse miettes existent, mais on peut distinguer deux grandes familles qui regroupent les gestionnaires avec marquage (*mark and sweep*) et les gestionnaires avec copie (*copy collector*).

Le principe du ramasse miettes *mark and sweep* est le suivant : l'adresse de chaque zone de mémoire allouée est mémorisée dans une table. Lorsqu'il n'y a plus assez de mémoire, toutes les données référencées par des variables du programmes, sont décorées pour indiquer qu'elles sont « vivantes » : c'est la phase de marquage. Dans une deuxième étape, la totalité des blocs de mémoire (le *tas*) est parcourue pour ne conserver que les zones contenant des objets «vivants», les autres étant rendues au système. Notons que la complexité de cet algorithme est proportionnelle à la taille du tas (les objets «vivants» sont marqués et les objets «morts» doivent être récupérés).

Le principe du ramasse miettes *copy collector* (“Stop and Copy” Using Semi-spaces) est différent : l'espace mémoire géré est divisé en deux demi-espaces de même taille. À un instant donné, un seul demi-espace est dit « actif » et toutes les allocations de mémoire se font dedans. Lorsque la mémoire devient insuffisante, tous les objets vivants dans ce demi-espace sont copiés dans l'autre demi-espace qui devient actif à son tour. Le demi-espace anciennement actif devenant

alors complètement libre. Cette approche est intéressante parce que le coût d'une allocation mémoire est très faible (du même ordre qu'une allocation effectuée dans une pile : il suffit de changer la valeur d'un pointeur indiquant la première zone libre du demi-espace actif) et l'efficacité dépend arbitrairement de l'espace mémoire disponible. La complexité de chaque phase de récupération est quant à elle proportionnelle à la taille des objets « vivants » en mémoire et non à la taille du tas. En contre partie, il faut deux fois plus de mémoire pour effectuer un même calcul. Il faut aussi noter que l'utilisation d'un tel ramasse miettes « déplace » les objets créés, ce qui nécessite l'utilisation d'un algorithme particulier, tel celui de Cheney (1970), pour copier les structures circulaires par exemple.

D'une manière générale, aucune des deux approches n'est meilleure que l'autre : elles ont chacune leurs avantages et leurs inconvénients. Le choix doit se faire en fonction du contexte d'utilisation, qui dépend de la mémoire totale disponible, de la taille des problèmes à traiter, de la durée de vie moyenne des objets créés et de la possibilité de les déplacer. Pour plus de détails concernant les différents types de ramasse miettes existant et leur comparaison, le lecteur est invité à se référer au livre de Jones et Lins (1996) ou au *survey* de Wilson (1992).

Si le principe de base d'un ramasse miettes est relativement simple, il faut savoir que la réalisation d'une implantation efficace est une tâche difficile et extrêmement technique. Lorsqu'on programme en `C` par exemple, il est généralement difficile, voire impossible de déterminer l'ensemble des objets vivants à un instant donné. En effet, il faut pour cela parcourir l'ensemble des variables utilisées par le programme, celles-ci étant mémorisées dans la pile système dont la structure dépend du processeur et du compilateur utilisé. De plus, les valeurs se trouvant dans la pile système ne sont pas typées et peuvent aussi bien correspondre à des pointeurs qu'à des entiers. Il existe des heuristiques permettant de différencier les pointeurs des entiers, mais lorsque la pile contient un entier ayant la même valeur qu'un pointeur, il devient impossible de savoir s'il s'agit d'un pointeur ou d'un entier. Dans le cadre d'un *mark and sweep*, ce n'est pas trop grave, puisqu'il peut supposer que c'est un pointeur et dans le pire des cas, ne pas rendre au système une zone qui aurait pu l'être. Mais dans le cadre d'un *copy collector*, les objets doivent être déplacés, ce qui entraîne une modification des valeurs se trouvant dans la pile. Cela peut devenir gênant lorsque la valeur d'un entier est modifiée. Il existe une solution permettant de traiter le cas où un entier a la même valeur qu'un pointeur (Bartlett 1988), mais je vous laisse imaginer la complexité de sa réalisation.

Il faut aussi savoir que l'efficacité d'un ramasse miettes ne dépend pas seulement de sa complexité théorique. Elle dépend grandement des choix d'implantations, liés à l'architecture de l'ordinateur, pour limiter les sauvegardes de registres, les « défauts de cache » et limiter la fragmentation de la mémoire. Elle dépend aussi d'heuristiques qui permettent de connaître le moment où une phase de récupération de mémoire doit être déclenchée. Récupérer trop souvent la mémoire amène à consacrer trop de temps au gestionnaire de mémoire, ce qui laisse moins de temps au programme pour effectuer ses calculs ; mais ne pas la récupérer assez souvent peut créer des phénomènes de « défaut de page » ou de défaut de cache qui ralentissent eux aussi la vitesse d'exécution du programme.

Dans le cadre de la réalisation du compilateur ELAN, l'objectif premier n'était pas d'implanter un nouveau ramasse miettes, d'autant plus qu'il existe une implantation disponible (Boehm et Weiser 1988), dont les performances sont à ce jour inégalées. Il s'agit d'un ramasse miettes à marquage dit « conservatif », qui pour les raisons mentionnées précédemment, peut ne pas rendre immédiatement au système certaines zones de mémoire qui ne sont effectivement plus utilisées. La pratique montre que la taille de la mémoire retenue est relativement constante, ce qui pénalise peu l'exécution des programmes. Dans un premier temps, nous avons choisi d'utiliser ce ramasse miettes et de nous concentrer sur d'autres aspects tels que la compilation du filtrage

associatif-commutatif par exemple. L'intégration du ramasse miettes (Boehm et Weiser 1988) nous a permis de simplifier considérablement la génération du code et d'améliorer d'environ 20% les performances des programmes générés par rapport à ceux qui utilisaient des compteurs de références. Mais depuis le début de cette nouvelle implantation du compilateur ELAN, de nombreuses améliorations ont été implantées, ce qui a pour effet de réduire la proportion de temps passée dans les étapes de filtrage et de gestion du non-déterminisme, par exemple, et d'augmenter celle passée dans le gestionnaire de mémoire (qui n'a pas été modifié). On peut ainsi trouver des exemples de programmes dont près de 50% du temps d'exécution est passé à gérer la mémoire. C'est principalement ce qui nous amène à étudier des techniques de ramasse miettes spécifiques, mieux adaptées à la programmation par réécriture.

Dans ce cadre, on peut remarquer qu'à chaque étape de réécriture, un grand nombre de symboles de petite taille sont alloués et que leur durée de vie moyenne est relativement courte. Lorsqu'on évalue une condition par exemple, le terme est construit, mis en forme normale puis détruit immédiatement après.

Cet aspect laisse ainsi penser qu'un ramasse miettes *copy collector* est bien adapté au cadre de la réécriture. D'un autre coté, l'utilisation de points de choix et l'application d'une stratégie *leftmost-innermost* font qu'un grand nombre de termes (les contextes) ne sont pas modifiés par l'application de règles sur des sous-termes, et dans ce cas l'utilisation d'un *mark and sweep* semble mieux adaptée. Ce type de situation n'est pas propre à ELAN et s'est présentée dans le cadre de l'implantation de langages fonctionnels tels que Caml (Cousineau et al. 1985, Weis et Leroy 1993, Cousineau et Mauny 1995, Leroy et Mauny 1993, Leroy 1995) par exemple. S'inspirant des travaux décrits dans (Doligez 1995, Doligez et Leroy 1993), nous pensons qu'une approche hybride utilisant deux ramasses miettes différents pourrait être avantageuse. L'idée consiste à définir un ramasse miettes à *générations* qui utilise un espace mémoire de taille fixe où l'allocation se fait linéairement (en modifiant la valeur d'un pointeur) à l'image du *copy collector*. Mais à la différence du gestionnaire à copie « classique », lorsque cet espace est plein, les termes vivants ne sont plus copiés dans un autre demi-espace, mais copiés dans une zone de mémoire gérée par un ramasse miettes *mark and sweep* tel que (Boehm et Weiser 1988) par exemple. L'intérêt d'un tel ramasse miettes, est que le coût des allocations les plus fréquentes devient constant (et presque nul). De plus, en supposant que le taux de mortalité des termes récemment alloués soit élevé, la phase de copie, proportionnelle aux nombres d'objets vivants, devient peu coûteuse. La taille de l'espace mémoire géré par le *mark and sweep*, quant à elle, devient alors nettement plus petite que celle gérée précédemment, ce qui augmente son efficacité.

Il faut noter que ce genre d'approche n'a d'intérêt que si un terme de « nouvelle génération » (géré par le *copy collector*) n'est jamais référencé par un terme « d'ancienne génération » (géré par le *mark and sweep*). Sinon il faudrait parcourir l'ensemble des termes d'ancienne génération pour déterminer et copier les termes de la nouvelle génération qui sont vivants, ce qui aurait une complexité comparable à celle d'un *mark and sweep* seul.

Dans le cadre d'ELAN, ce genre de situation n'arrive jamais lorsqu'on s'interdit de réutiliser des morceaux du membre gauche pour construire le terme réduit. En effet, la construction du bas vers le haut (*bottom-up*) d'un terme garantit que les sous-termes d'un symbole sont « plus vieux » que le symbole ; il n'y a donc jamais de référence vers un terme se trouvant dans une génération plus jeune.

Le concept de base des ramasse miettes à générations est de supposer qu'un objet qui a survécu longtemps a de grandes chances de vivre encore longtemps. Et en fonction du comportement moyen des objets alloués, il est possible d'ajuster la structure du ramasse miettes en ajoutant des générations. Il est aussi possible de modifier la stratégie interne du ramasse miettes qui fait passer un objet d'une génération vers une autre. Dans le cadre d'ELAN, nous envisa-

geons d'implanter une première version utilisant deux générations, et d'adapter la structure du ramasse miettes en fonction des résultats expérimentaux.

11.5 Synthèse

Dans ce chapitre, nous avons présenté sommairement les structures de données utilisées par les programmes engendrés par le compilateur ELAN. Nous avons aussi mis en lumière différents problèmes liés à la gestion de la mémoire et à l'intégration de sortes et d'opérations *builtins*. Mais ces travaux ne sont que « la partie visible de l'iceberg ». En effet, le développement de cette bibliothèque de support et la volonté constante de définir des constructions efficaces et réutilisables nous a donné cette expériencre, difficilement transmissible, qui donne l'intuition et le recul nécessaire à tout développement logiciel de qualité.

Chapitre 12

Expériences pratiques

12.1 Estimation du degré de compilation 166
12.2 Évaluation des performances 172
12.3 Coût du filtrage AC 176
12.4 Comparaison avec d'autres implantations 178

La réalisation et la diffusion d'un compilateur passe par de nombreuses phases d'expérimentation et d'évaluation. Il est en particulier intéressant de connaître le comportement des programmes générés en terme de fiabilité, d'efficacité et de consommation mémoire.

Dans ce chapitre nous nous proposons d'évaluer les méthodes de compilation proposées précédemment en nous concentrant essentiellement sur trois aspects :

- quels sont les apports du compilateur dans le cadre du projet ELAN ? Pour y répondre, nous comparons les performances du compilateur actuel avec celles de l'interpréteur ELAN ;
- dans le cadre de la réécriture modulo AC, quel est le comportement des programmes générés et quelles sont les étapes du processus de normalisation qu'il serait intéressant d'améliorer ? Nous étudions la proportion de temps passé dans les différentes étapes qui composent le processus de normalisation AC ;
- comment se situe le compilateur ELAN par rapport aux autres implantations de langages à base de règles de réécriture ? Nous le comparons, sur un échantillon de programmes, avec d'autres systèmes implantant une procédure de normalisation AC, et plus particulièrement avec Brute et Maude, qui sont deux excellentes implantations.

La réalisation de *benchmarks* est une tâche souvent difficile parce qu'elle consiste à généraliser des résultats obtenus à partir d'un petit nombre d'expériences et ceci, généralement sans utiliser de méthodes statistiques telles que l'analyse de séries chronologiques. Comme le mentionne Bailey (1995) dans sa thèse, il semble y avoir deux « philosophies » d'évaluation des performances d'un compilateur :

- l'évaluation des « petits » programmes, facilement compréhensibles et expérimentables dans différents langages ou sur différentes architectures. Le programme qui calcul le $n^{\text{ième}}$ nombre de Fibonacci est un exemple typique de « petit » programme qui met en évidence, de façon extrême, certains comportements tels que le traitement des entiers et de la récursivité. L'inconvénient de ce type de programmes est qu'ils ne reflètent que partiellement les capacités d'une implantation en ne testant qu'un sous-ensemble restreint des constructions du

langage source. L'avantage est que ces programmes sont facilement portables et expérimentables avec d'autres compilateurs, ce qui permet de se situer, même approximativement, par rapport aux autres ;

– l'évaluation de « gros » exemples donne une image généralement plus réaliste des performances moyennes d'un compilateur. La taille et la spécificité des programmes rend alors plus difficile la comparaison avec d'autres outils de compilation. Lorsqu'on dispose d'un interpréteur et d'un compilateur, ce type d'expérimentation permet néanmoins de mesurer les apports et de mettre en valeur l'intérêt des nouvelles méthodes développées.

Pour évaluer les qualités du compilateurs ELAN, nous utilisons ces deux types d'approches : des « petits » exemples pour le situer par rapport aux autres implantations et mettre en évidence les caractéristiques de l'algorithme de filtrage AC proposé ; de « gros » programmes pour montrer sa capacité à traiter des cas réels et pour caractériser plus précisément les apports de cette thèse.

12.1 Estimation du degré de compilation

Dans le chapitre 4 (page 58), nous avons présenté deux grandes approches pour compiler un langage donné : la première consistant à représenter les structures dominantes du langage source par des structures de *données* du langage cible[5], la deuxième consistant à représenter les caractéristiques du langage source par des structures de *contrôle* du langage cible.

Dans cette partie, nous proposons de comparer le comportement des programmes compilés et interprétés en retenant comme indicateurs la quantité totale de mémoire allouée et le nombre total de symboles de fonction construits au cours d'un calcul. Ces informations nous donnent une estimation du degré de traduction des structures du langage source en structures de *contrôle* du langage cible. Les résultats expérimentaux suivants montrent que pour effectuer un même calcul, le compilateur alloue moins de mémoire et crée moins de symboles de fonction que l'interpréteur. Il est évidemment intéressant d'observer ces diminutions d'allocation et de création, mais il l'est encore plus de comparer ces diminutions entre elles. Nous appelons « degré de compilation » le rapport entre la diminution de mémoire allouée et la diminution du nombre de symboles construits.

12.1.1 Fib_builtin

C'est probablement un des *benchmarks* les plus fréquents en programmation fonctionnelle, il permet d'évaluer l'efficacité des appels récursifs et les opérations *builtins* sur les entiers. Le programme ELAN s'exprime en trois règles de réécriture :

```
[] fib(0) => 1                              end
[] fib(1) => 1                              end
[] fib(n) => fib(n-1) + fib(n-2) if n>1 end
```

Les comportements de l'interpréteur et du compilateur sont illustrés sur la figure 12.1. Nous nous trouvons ici dans une situation extrême où le compilateur ne fait quasiment aucune allocation dynamique de mémoire : aucun symbole de fonction n'est construit au cours de l'exécution du programme compilé[6]. L'interpréteur a quant à lui un comportement différent : les termes de sorte `builtinInt` sont représentés en utilisant un opérateur « d'emballage » présenté dans

5. Celles-ci étant ensuite évaluées par un interprète intégré au code généré.

6. La figure de droite illustre la création d'un symbole de fonction, ceci parce que la valeur 0 se représente difficilement sur un graphique en échelle logarithmique.

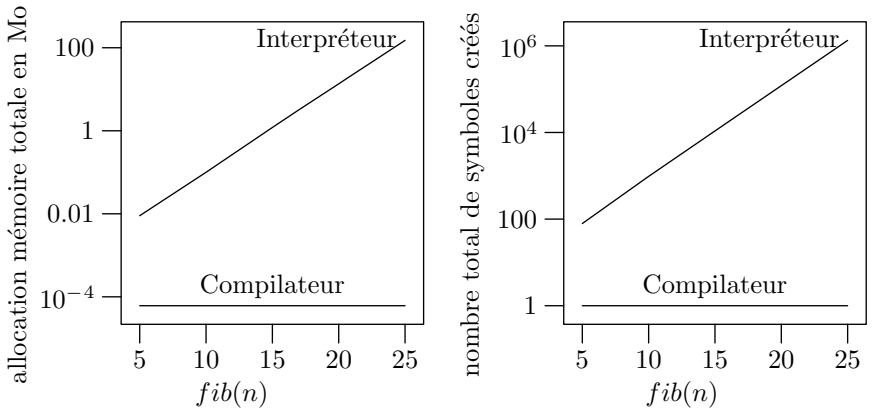


FIG. 12.1 – *Degré de compilation de Fib_builtin*

le chapitre 11, ce qui l'oblige à allouer de la mémoire et à créer un symbole de fonction après chaque étape de réécriture.

12.1.2 NqueensAC

C'est là aussi un des *benchmarks* les plus utilisés en programmation logique, il permet d'évaluer les capacités du langage à gérer efficacement la pose et la gestion de points de choix. Le problème consiste à trouver toutes les façons de placer n reines sur un échiquier de taille $n \times n$, de telle sorte qu'aucune d'elles ne soit mise en échec par une autre.

Nous présentons ici un codage plus concis, plus efficace et surtout plus élégant que celui présenté dans le chapitre 8. L'idée consiste à utiliser un opérateur d'union AC pour représenter les ensembles de positions qu'il est possible d'associer à une reine. La signature comporte les 7 opérateurs suivants :

```
queens(@,@) : (set list[int])     list[int];
ok(@,@,@)   : (int int list[int]) bool;
@ U @       : (set set)           set (AC);
(@)         : (int)               set;
[@ U @]     : (int set)           set;
Set(@)      : (int)               set;
Empty       :                     set;
```

Comme précédemment, le prédicat `ok(@,@,@)` est vrai lorsqu'une reine nouvellement placée n'est pas mise en échec par les autres reines se trouvant déjà sur l'échiquier. Sa définition est la suivante :

```
rules for bool
  p, d, diff : int;
  l          : list[int];
global
  [] ok(diff,d,nil)     => true                          end
  [] ok(diff,d,p.l)     => false        if d-p == diff   end
  [] ok(diff,d,p.l)     => false        if p-d == diff   end
  [] ok(diff,d,p.l)     => ok(diff+1,d,l)                end
end
```

La stratégie et les trois règles suivantes permettent quant à elles de créer l'ensemble $(0) \cup (1) \cup \cdots \cup (n)$ et d'en extraire les éléments un à un.

```
rules for set
  S : set;
  i : int;
global
  []            Set(0)  => Empty U (0)    end
  []            Set(i)  => Set(i-1) U (i) end
  [extractrule] (i) U S => [i U S]        end
end

strategies for set
  [] extractPos => dk(extractrule) end
end
```

Il faut noter que la règle `extractrule` s'applique sur un ensemble S' et sélectionne (par filtrage AC) un élément (i) et un ensemble S tels que $S \cup (i) = S'$. L'application de la règle permet de construire le terme `[i U S]` (dans ce dernier cas, le symbole `[@U@] : (int set) set` n'est pas AC). Avec ce codage particulier, le programme de résolution du problème des n reines s'exprime avec seulement deux règles de réécriture et une stratégie :

```
rules for list[int]
  pos     : int;
  S,reste : set;
  l       : list[int];
local
   [queensrule]  queens(S,l)     => queens(reste,pos.l)
                 where (set) [pos U reste] :=(extractPos) S
                 if ok(1,pos,l)
   end
   [final]       queens(Empty,l) => l end
end

strategies for list[int]
  [] queens => repeat*(dk(queensrule)); dc(final) end
end
```

La règle `queensrule` s'applique récursivement en cherchant à chaque étape une position

satisfaisant le prédicat `ok`. La règle `final` permet d'arrêter la récursion une fois que l'ensemble des positions à étudier est vide. Cet exemple illustre une fois encore la présence d'un double non-déterminisme lorsque des symboles AC sont utilisés dans une règle conditionnelle appliquée avec une stratégie d'exploration de type `dk`.

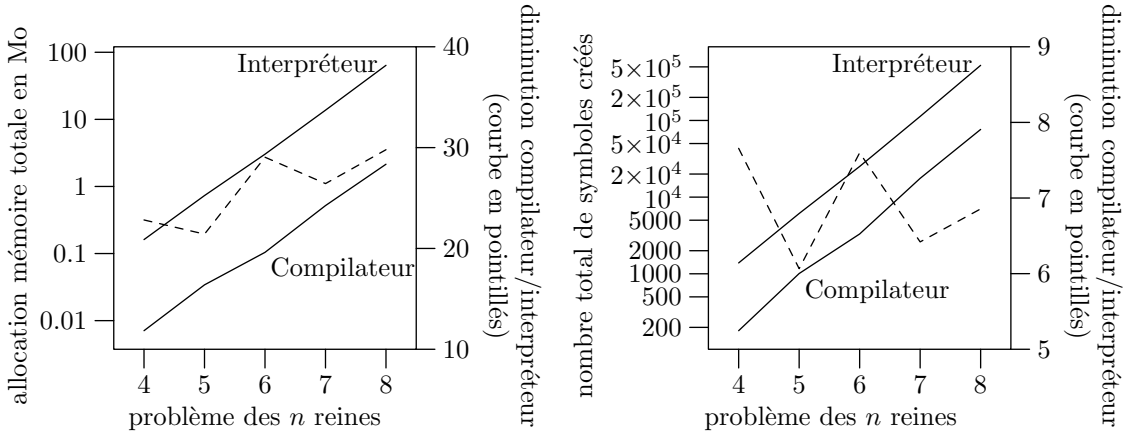


FIG. 12.2 – *Degré de compilation de NqueensAC*

Les résultats expérimentaux présentés sur la figure 12.2 sont intéressants parce qu'ils montrent, dans un cas relativement complexe, une très nette diminution du nombre total d'octets et de symboles alloués lorsque le compilateur est utilisé. Le graphique de gauche comptabilise, en fonction de la taille du problème à résoudre, le nombre total d'octets alloués dynamiquement par une fonction de type `malloc`. Il est important de s'assurer ici que ce nombre n'est pas la quantité de mémoire consommée à un moment donné, mais bien la somme cumulée des tailles des zones mémoires demandées pour résoudre le problème. De même, le graphique de droite n'illustre pas le nombre de symboles « vivants » à un moment donné mais bien le nombre de symboles créés au cours de l'exécution.

Notons que le compilateur permet de diviser par 7 environ le nombre de symboles créés (courbe en pointillés) : cela vient en partie du fait que la construction des symboles définis est traduite par des appels de fonctions dans le code généré, alors que l'interpréteur est obligé de construire effectivement ces symboles. Si l'on compare cette diminution par 7 du nombre de symboles créés avec la taille totale de mémoire allouée, qui est environ 25 fois plus petit lorsque le compilateur est utilisé, on est en droit de se demander pourquoi un tel écart. On pourrait dans un premier temps penser que la taille d'un symbole de l'interpréteur est 4 fois supérieure à celle d'un symbole du compilateur, mais ce n'est pas le cas : il y a certes une différence liée à la représentation des termes et aux techniques de ramasse miettes utilisées, mais la différence de taille entre les deux types de représentation est inférieure à 2. Le facteur 2 restant vient donc du fait que bon nombre de constructions du langage source (ELAN), telles que les conditions ou les stratégies, étaient représentées par des structures de données de l'interpréteur alors qu'elles sont maintenant traduites en structures de contrôle du langage cible (le `C`) qui n'entraînent pas d'allocation dynamique de mémoire.

Ces informations nous montrent que le compilateur ne se contente pas de réduire la taille des objets créés mais qu'il effectue un travail en profondeur, lié à la structure du code généré, qui permet de réduire le nombre d'allocations dynamiques de mémoire.

12.1.3 ANS-Complétion

Cet exemple correspond à l'implantation d'une procédure de complétion définie dans (Lescanne 1989) et étendue dans (Moreau 1994, Kirchner et Moreau 1995). Ce *benchmark* fait partie de la catégorie des « gros » programmes parce qu'il se compose de plusieurs centaines de règles et de quelques dizaines de stratégies. Le système p_n à « compléter » est une variante de l'axiomatisation des groupes qui comportent n éléments neutres et n opérateurs inverses :

$$p_n = \left\{ \begin{array}{ccc} f(f(x,y),z) & = & f(x,f(y,z)) \\ f(e_1,x) & = & x \\ \vdots & \vdots & \vdots \\ f(e_n,x) & = & x \\ f(x,i_1(x)) & = & e_1 \\ \vdots & \vdots & \vdots \\ f(x,i_n(x)) & = & e_n \end{array} \right.$$

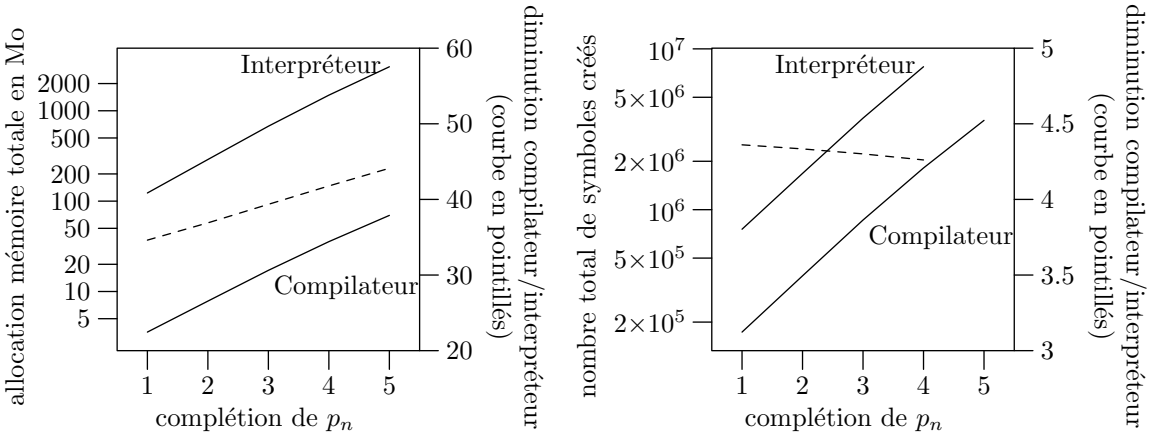


FIG. 12.3 – *Degré de compilation de ANS-Complétion*

La figure 12.3 montre que, pour l'exemple de la complétion, le « degré de compilation » est de l'ordre de 10 puisque la diminution du nombre d'allocations de symboles de fonction est environ dix fois moins importante que la diminution du nombre total de mémoire allouée.

12.1.4 Bool3

C'est un *benchmark* (imaginé par Steven Eker) qui définit un système de calcul dans une logique à 3 valeurs et qui permet d'évaluer les performances des procédures de normalisation AC. Le système de réécriture comporte les règles suivantes, où $+$ et $*$ sont des opérateurs AC :

$x+0$	$\to$	x	$x*0$	$\to$	0
$x+x+x$	$\to$	0	$x*x*x$	$\to$	x
$(x+y)*z$	$\to$	$(x*z)+(y*z)$	$x*1$	$\to$	x
$and(x,y)$	$\to$	$(x*x*y*y)+(2*x*x*y)+$			
		$(2*x*y*y)+(2*x*y)$			
$or(x,y)$	$\to$	$(2*x*x*y*y)+(x*x*y)+$			
		$(x*y*y)+(x*y)+(x+y)$			
$not(x)$	$\to$	$(2*x)+1$			
2	$\to$	$1+1$			

Le *benchmark* consister à normaliser les deux termes suivants et à comparer leur forme normale :

$$and(and(a_1,a_2),\ldots,and(a_{n-1},a_n))$$

et

$$not(or(or(not(a_1),not(a_2)),\ldots,or(not(a_{n-1}),not(a_n))))$$

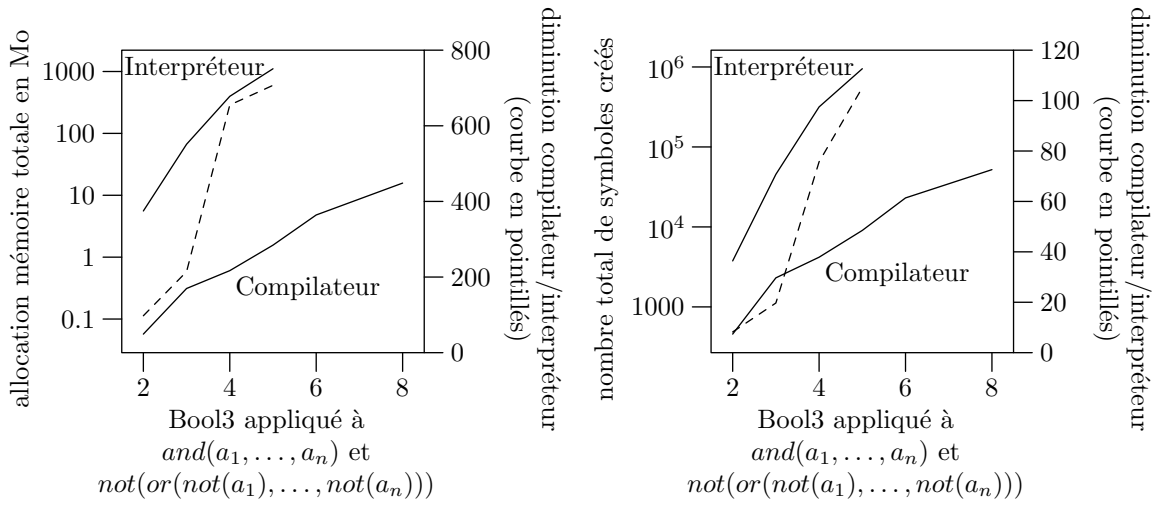


FIG. 12.4 – *Degré de compilation de Bool3*

La figure 12.4 montre une diminution très importante de la taille mémoire et du nombre de symboles alloués. Cette diminution est d'autant plus intéressante qu'elle s'accroît en fonction de la complexité du problème à résoudre. On peut aussi remarquer que les deux courbes en pointillés sont sensiblement les mêmes : la diminution de mémoire est environ 8 fois plus importante que la diminution du nombre de symboles créés. Ce facteur, à comparer avec 4 pour le problème des n reines et 10 dans le cas de la complétion de Knuth-Bendix, semble montrer que le « degré de compilation » reste « normal » lorsque que des règles avec symboles AC sont compilées. On aurait pu craindre une nette diminution de ce « degré de compilation » dans la mesure où la nature des problèmes de filtrage AC fait que de nombreuses structures de données doivent être

créées dynamiquement. Ces résultats sont sûrement liés au fait que nous avons tenté de réduire au minimum ces allocations dynamiques en compilant des automates de filtrages et des fonctions d'accès, en imposant des restrictions sur la structure des termes et en utilisant des structures de données compactes. Évidemment, pour conserver un cadre suffisamment général, nous ne pouvons pas échapper à la construction dynamique de graphes bipartis, de substitutions et à la maintenance en forme canonique des termes manipulés. Ce sont ces caractéristiques qui rendent le processus de normalisation AC difficile à compiler, au sens : « génération exclusive de structures de contrôle ».

12.2 Évaluation des performances

Dans cette partie, notre objectif est d'évaluer les apports du compilateur en terme de puissance et de performance. Il est clair que comparer le compilateur avec l'interpréteur ELAN ne permet pas d'évaluer, dans l'absolu, la qualité des algorithmes proposés et leur implantation. Cela permet néanmoins de mesurer le chemin parcouru depuis le début de cette thèse.

ELAN n'est ni Fortran, ni C, ni Java et n'est pas une solution à tous les problèmes, il est cependant intensivement utilisé par notre équipe et d'autres groupes de recherche travaillant en déduction automatique, en résolution de contraintes et sur les langages de spécification algébrique. Divers algorithmes, tels que des procédures d'unification d'ordre supérieur, des outils de preuve de terminaison ou des résolveurs de contraintes ont été spécifiés en ELAN et ont montré les limites de notre interpréteur, bien que celui-ci fasse partie des « bons» interpréteurs, au même titre qu'ASF+SDF ou OBJ.

En illustrant l'apport réel des techniques de compilation développées dans cette thèse, nous espérons montrer que celles-ci permettent de développer des applications et résoudre des problèmes qui n'auraient pas pu l'être sans son existence.

12.2.1 Fib_builtin et NqueensAC

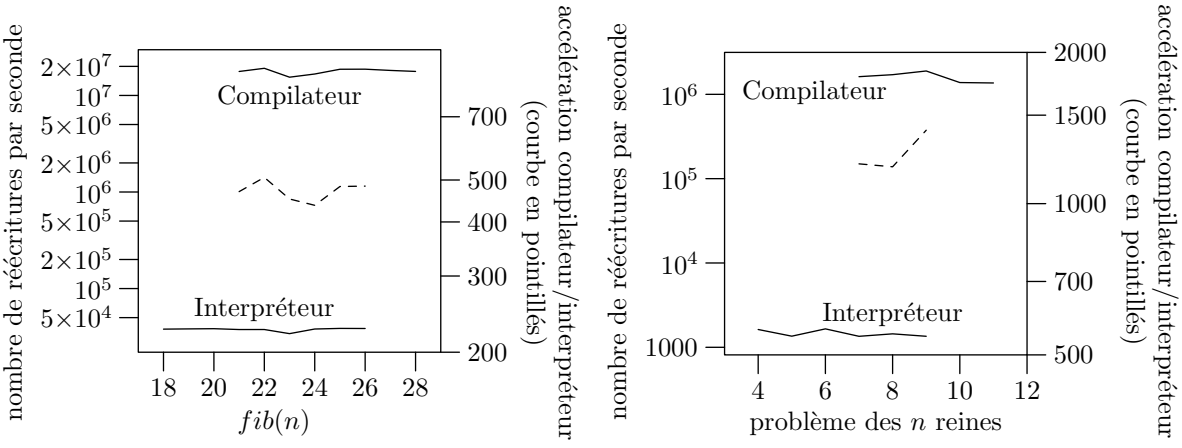


FIG. 12.5 – *Évaluation des performances de Fib_builtin et NqueensAC*

Comme le montre la partie gauche de la figure 12.5, dans un cadre purement fonctionnel, en présence de calculs arithmétiques intensifs, le compilateur permet d'atteindre de « bonnes » performances où plus de 17 millions de règles sont appliquées chaque seconde. L'accélération par rapport au compilateur est comprise en 400 et 500, ce qui signifie qu'une heure de calcul avec le

compilateur correspond à 2 ou 3 semaines de calcul pour l'interpréteur. Toutes ces mesures ont été obtenues sur une station Dec Alpha 500.

Le graphique de droite montre que dans un cadre complètement non-déterministe, l'accélération offerte par le compilateur est encore plus importante puisqu'elle est supérieure à 1000.

12.2.2 ANS-Complétion

Cet exemple plus complexe exploite une grande partie des constructions du langage ELAN et en particulier les règles conditionnelles et les stratégies.

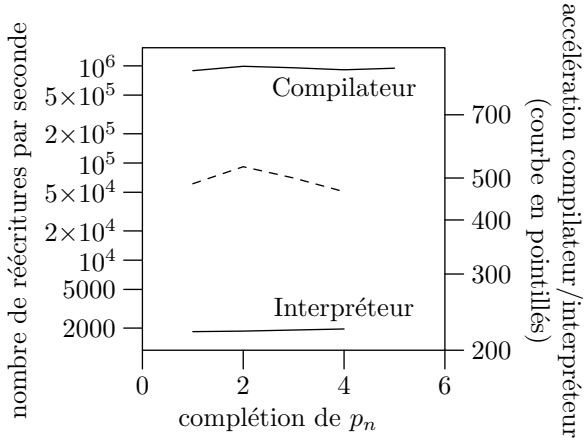


FIG. 12.6 – *Évaluation des performances de ANS-Complétion*

La figure 12.6 montre que dans le cadre d'une application « réelle » telle que la procédure de complétion de Knuth-Bendix (1970), le compilateur permet d'appliquer près d'un million de règles par seconde, ce qui le rend, ici encore, entre 400 et 500 fois plus rapide que l'interpréteur. La complétion du système p_8 mettant environ 24 secondes, je vous laisse calculer le temps qu'il fallait attendre pour résoudre ce même problème lorsque seul l'interpréteur était disponible.

12.2.3 Bool3, Set et Nat10

L'exemple Bool3 est intéressant parce qu'il montre (figure 12.7) dans un cadre extrême de normalisation AC, que les performances du compilateur se dégradent nettement moins que celles de l'interpréteur lorsque la taille des termes manipulés pour résoudre un problème augmente.

Le programme Set correspond à un système de réécriture permettant de manipuler des ensembles et d'effectuer des opérations telles que le calcul de l'ensemble des parties d'un ensemble donné (voir annexe A.4). Le *benchmark* présenté sur la figure 12.8 consiste à calculer la cardinalité de l'ensemble $\mathcal{P}(\{1,\ldots,n\})$

L'exemple Nat10 correspond à un système de réécriture modulo AC présenté dans (Contejean, Marché et Rabehasaina 1997). Ce système permet d'effectuer des calculs arithmétiques sur les entiers et a pour particularité d'utiliser 56 règles commençant par le symbole AC $+$, 11 règles commençant par le symbole AC $*$ et 82 règles syntaxiques. À l'époque, les auteurs conjecturaient dans leur article que des techniques de compilations *many-to-one* devraient permettre d'améliorer les performances de ce système[7].

7. Cet exemple était originellement implanté en C*i*ME, qui est plus un prouveur automatique qu'un outil de

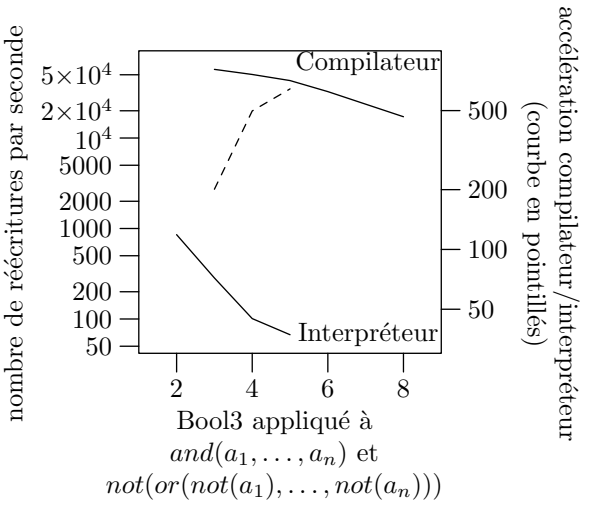


FIG. 12.7 – *Évaluation des performances de Bool3*

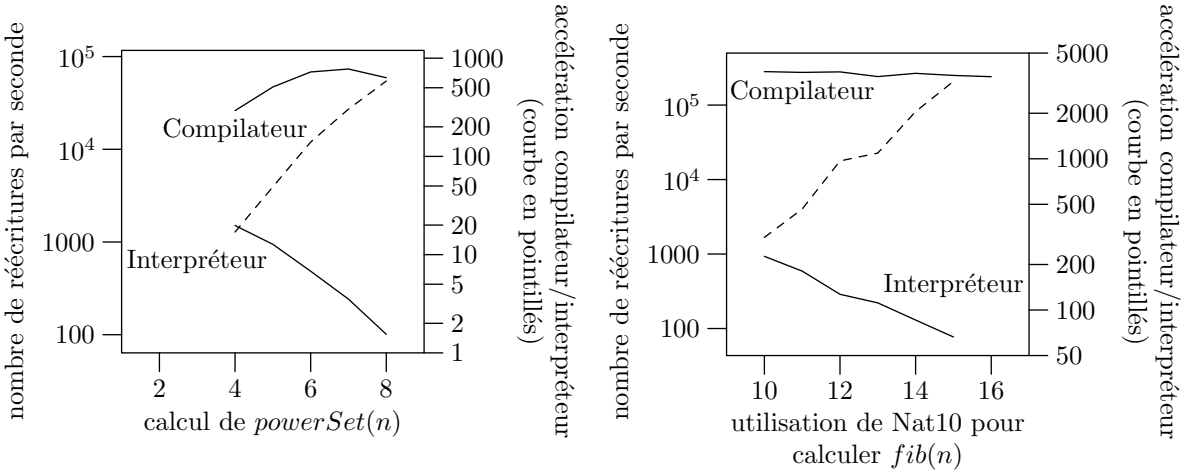


FIG. 12.8 – *Évaluation des performances de Set et Nat10*

Les résultats présentés sur les figures 12.7 et 12.8 montrent clairement l'intérêt des techniques de compilation de la normalisation AC : l'accélération offerte par le compilateur est souvent supérieure à 200. Notons aussi que cette accélération augmente en fonction de la taille du problème à résoudre.

12.2.4 Minela

Cette exemple, présenté dans le chapitre 9, correspond à l'implantation d'un méta-interpréteur ELAN écrit en ELAN dont le fonctionnement est décrit dans (Kirchner et Moreau 1996).

Les résultats de la figure 12.9 illustrent l'utilisation de ce méta-interpréteur pour calculer les n premiers nombres premiers ainsi que le terme de preuve associé au calcul. Ici, l'accélération

normalisation. Pour calculer le 16$^{\text{ième}}$ nombre de Fibonacci, la version 1.3 de C*i*ME appliquait 10,599 règles de réécriture en 4h30. La nouvelle version résout maintenant le problème en approximativement 5 minutes, sur une même architecture.

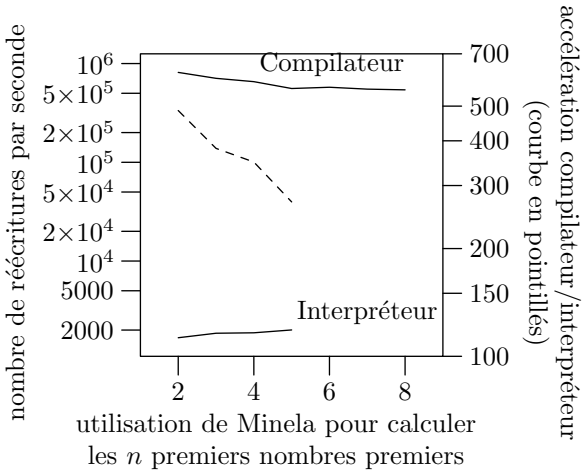


FIG. 12.9 – *Évaluation des performances de Minela*

semble décroître, mais cela vient en partie de mesures imprécises liées à des temps d'exécution trop petits des programmes compilés. En augmentant la taille des problèmes, l'accélération devrait se stabiliser entre 200 et 300, mais l'interpréteur met trop de temps pour que nous puissions effectuer ces mesures.

12.2.5 Applications de taille « réelle »

Les résultats précédents montrent que les techniques de compilation développées permettent d'accroître considérablement la vitesse d'exécution des spécifications ELAN tout en réduisant leur consommation mémoire. Ces travaux ont en particulier permis de développer deux applications majeures, pour lesquelles l'interpréteur seul n'aurait pas suffit : il s'agit de Colette et d'une bibliothèque de calcul sur les automates d'arbres.

Colette. C'est un environnement de résolution de contraintes développé par Carlos Castro dans le cadre de sa thèse (1998). À titre d'exemple, pour un problème de résolution de contraintes lié à un problème de conversion de calendriers, les caractéristiques de la spécification sont les suivantes[8] : 73 définitions de modules, 42 définitions de sortes, 969 règles de réécriture (dont 348 nommées), 46 conditions, 473 évaluations locales du type `where` et 170 évaluations locales du type `choose/try`, 63 stratégies se composant de 11 iterate*, 14 repeat*, 67 dc one, 31 dc, 16 dk et 20 compositions de stratégies (`;`). Sur ce type d'application, le compilateur applique environ 300.000 règles de réécriture par seconde, et ceci pour des calculs pouvant durer plusieurs jours. Pour certains problèmes d'ordonnancement, il nous est ainsi arrivé de dépasser la dizaine de milliards de règles appliquées, ce qui correspond à près de six mois de calcul avec l'interpréteur.

Calcul sur les automates d'arbres. C'est un environnement de calcul et de preuve développé par Thomas Genet dans le cadre de sa thèse (1998). Cette bibliothèque est actuellement utilisée au CNET pour montrer automatiquement des propriétés de confidentialité et d'authentification de protocoles cryptographiques nouvellement développés. Les caractéristiques des spécifications

8. Ces données sont approximatives parce qu'une partie de la spécification est engendrée automatiquement en fonction du problème à résoudre.

sont par exemple celles-ci : 105 définitions de modules, 107 définitions de sortes, 824 règles de réécriture (dont 298 nommées), 170 conditions, 185 évaluations locales du type `where` et 17 évaluations locales du type `choose/try`, 55 stratégies se composant de 19 iterate*, 33 repeat*, 61 dc one, 65 dc, 6 dk et 24 compositions de stratégies (;).

Sur cet exemple, le compilateur a appliqué plus de 524 millions de règles en moins de 10 minutes, ce qui aurait pris près de quatre jours à l'interpréteur pour effectuer le même calcul.

12.3 Coût du filtrage AC

Depuis les travaux de compilation de Michael J. O'Donnell et de Robert Strandh, nous savons que l'étape la plus coûteuse d'une procédure de normalisation syntaxique n'est pas le filtrage mais bien la construction du terme réduit. Par contre, dans le cas associatif et commutatif, nous étions encore très loin de ces conclusions. Avant d'étudier les différents moyens de compiler la réécriture modulo AC, nous avions effectué des expérimentations avec l'interpréteur ELAN afin d'évaluer l'intérêt potentiel de telles méthodes de compilation. En analysant la répartition des calculs de l'interpréteur sur des exemples tels que Bool3, Nat10 ou Somme, nous avions remarqué que plus de 80% du temps de calcul était consacré au filtrage AC, ce qui montrait clairement l'intérêt d'améliorer l'efficacité d'une telle procédure.

Pour ces trois programmes, le tableau suivant donne un aperçu du temps passé dans les opérations liées au filtrage AC[9] lorsque le compilateur est utilisé. Le total de chaque colonne n'est pas égal à 100% parce que d'autres fonctions, non prises en compte ici, sont impliquées dans le processus de normalisation et aussi parce que certaines fonctions sont comptées plusieurs fois : les deux lignes du bas indiquent le temps total passé dans la gestion de la mémoire et la gestion du non-déterminisme, mais ceux-ci ont déjà été comptabilisés dans la construction des graphes bipartis compacts (CBG) et la résolution des graphes bipartis par exemple.

	Bool3	Nat10	Somme
construction des CBG	12.8%	31.77%	8.24%
extraction des graphes bipartis	0.45%	4.39%	0.11%
résolution des graphes bipartis	1.57%	2.04%	5.13%
Total filtrage AC	14.82%	38.20%	13.48%
construction des substitutions	4.01%	5.45%	15.78%
maintenance des formes canoniques	21.9%	3.74%	0.41%
gestion de la mémoire	29.6%	27.06%	5.93%
gestion du non-déterminisme	3.83%	4.81%	49.15%

Bool3. En analysant cet exemple, composé d'un petit nombre de règles (dont l'application engendre de très gros termes), on s'aperçoit que le temps passé dans la construction et la résolution des graphes bipartis est inférieur à 15% du temps total d'exécution. On peut noter le coût relativement faible de la construction des substitutions, ce qui montre l'intérêt de notre approche consistant à compiler des fonctions d'accès. En revanche, on peut estimer que le temps passé à maintenir les termes en forme canonique est relativement important, mais c'est principale-

9. Ces données ont été obtenues sur un Sun Ultrasparc en utilisant l'utilitaire `quantify`.

ment dû à la non-linéarité droite du système de réécriture qui entraîne de nombreuse étapes d'aplatissement et de fusion de listes triées.

Nat10. Cet exemple est particulier parce que les termes manipulés sont petits, mais le nombre de règles composant le système est important (plus de 50 commençant par le symbole AC +). Le temps passé à construire des graphes bipartis compacts occupe près de 30% du temps total, mais une analyse plus fine nous montre que seulement 3% de ce temps est passé dans le filtrage des sous-termes et que 80% du temps restant consiste à allouer de la mémoire pour mémoriser les « 50 » vecteurs de bits servant à représenter les arêtes du graphe. Il faut bien voir que ces graphes bipartis sont alloués et détruits au début et à la fin de chaque application d'une règle commençant par un symbole AC et c'est pourquoi nous sommes optimiste : nous envisageons de mettre en place un nouveau mécanisme de gestion mémoire capable de « recycler » les structures de données pour pouvoir les utiliser d'une application de règle à un autre. Nous pensons ainsi réduire considérablement le temps passé dans la construction des graphes bipartis compacts, ce qui améliorerait les performances générales.

Les autres données nous montrent que le temps passé à extraire et résoudre les graphes bipartis est relativement petit et qu'une fois encore, c'est le mécanisme de gestion mémoire qui doit être amélioré en priorité en utilisant les méthodes proposées dans le chapitre 11 par exemple.

Somme. Ce dernier exemple utilise un opérateur AC d'union ($\cup$) et trois règles conditionnelles pour extraire des entiers d'un ensemble et calculer leur somme ($\Sigma_{i=1}^{100} i$). Le système de réécriture est défini de la manière suivante :

$x \in \emptyset$	$\rightarrow$	$\bot$
$x \in s$	$\rightarrow$	$\mathsf{check}(x \in' s)$
$x \in' s \cup \mathsf{set}(y)$	$\rightarrow$	$\top$ **if** $x = y$
$\mathsf{check}(\top)$	$\rightarrow$	$\top$
$\mathsf{check}(x \in' s)$	$\rightarrow$	$\bot$
$\mathsf{state}(s_1 \cup \mathsf{set}(x), s_2, y)$	$\rightarrow$	error **if** $x \in s_2$
$\mathsf{state}(s_1 \cup \mathsf{set}(x), s_2, y)$	$\rightarrow$	$(\mathsf{state}(s_1, s_2 \cup \mathsf{set}(x), x + y)$ **if** $x \notin s_2$

Le *benchmark* consiste à normaliser le terme :

$$\mathsf{state}(\emptyset \cup \mathsf{set}(1) \cup \cdots \cup \mathsf{set}(100), \emptyset, 0)$$

Le résultat attendu étant :

$$\mathsf{state}(\emptyset, \emptyset \cup \mathsf{set}(1) \cup \cdots \cup \mathsf{set}(100), 5050)$$

Lorsqu'une stratégie *leftmost-innermost* est appliquée, ce système est particulièrement intéressant parce qu'il teste la capacité des algorithmes de filtrage AC à extraire non plus une solution mais toutes les solutions d'un problème donné. Dans le cadre de notre *benchmark*, notons que la règle $\mathsf{state}(s_1 \cup \mathsf{set}(x), s_2, y) \rightarrow \mathsf{error}$ **if** $x \in s_2$ ne s'applique jamais parce que la condition **if** $x \in s_2$ n'est jamais satisfaisable, mais pour le savoir, le système doit calculer toutes les instances possibles de la variable x et vérifier qu'elles n'appartiennent pas à s_2.

Il est ainsi naturel de voir la proportion de temps passé dans la gestion du non-déterminisme augmenter par rapport aux précédents exemples, mais nous estimons qu'elle est ici excessive et

nous envisageons de modifier légèrement nos schémas de compilation pour réduire la taille des environnements à sauvegarder lors de la pose des points de choix : il suffit pour cela de favoriser l'utilisation de variables globales afin de réduire le nombre de variables locales nécessaires dans les fonctions `C` générées.

En revanche, on peut constater que le temps passé dans les fonctions de filtrage AC reste relativement petit. Le coût lié à la construction des substitutions est supérieur à celui des exemples précédents, mais c'est un comportement normal étant donné le nombre de substitutions calculées et construites pour tester la satisfaisabilité de la condition **if** $x \in s_2$. Même si la proportion de temps passé à construire les substitutions peut paraître importante, là encore, la compilation des fonctions d'accès permet de limiter cette augmentation.

12.4 Comparaison avec d'autres implantations

Comparer rigoureusement différentes implantations d'un même langage de programmation est une tâche déjà bien difficile. Classer, en fonction de leurs performances, différentes implantations de langages différents, est quasiment impossible. Il y a d'une part l'effet *benchmarks* qui fausse les mesures, simplement parce que les concepteurs sont amenés à optimiser les algorithmes les plus utilisés par ces ensembles de programmes. Et d'autre part, s'ajoute la difficulté de choisir les *benchmarks*, sachant que certains problèmes s'expriment mieux dans un langage plutôt que dans un autre.

Dans cette partie, nous nous contentons de situer ELAN par rapport aux autres outils permettant d'effectuer de la réécriture modulo AC, et nous nous concentrons particulièrement sur les comparaisons avec Brute et Maude, qui font partie des meilleurs « moteurs de réécriture » diffusés à ce jour[10]. Notre objectif n'est pas d'établir un classement « rigoureux » entre ces trois implantations, mais plutôt d'illustrer l'intérêt des méthodes développées dans les chapitres précédents et de montrer que leur implantation peut être largement compétitive avec les autres, même si dans notre situation, la marge d'amélioration de certains algorithmes techniques est encore importante.

12.4.1 Calculs déterministes

La figure 12.10 illustre les performances du compilateur dans un cadre fonctionnel où seul le filtrage syntaxique est utilisé. Tous les temps donnés dans cette partie ont été obtenus sur une station Sun Ultra 1.

La figure de gauche montre que dans le cadre de calculs arithmétiques intensifs, les techniques de compilation permettent d'améliorer considérablement l'efficacité des programmes : un facteur 100 sépare encore ELAN des meilleurs interpréteurs[11]. La courbe `ELAN-natc10` correspond aussi aux calculs de $fib(n)$, mais effectués en utilisant ELAN et une variante du système Nat10 présenté précédemment : Nat10 utilise des opérateurs AC ($+_{AC}$ et $*_{AC}$) et des règles de la forme $g(x) +_{AC} h(y) \to r(x,y)$. Dans cette situation, l'opérateur $+_{AC}$ n'a alors besoin d'être que commutatif et il nous suffit de dupliquer ces règles pour en dériver un système n'utilisant que des opérateurs syntaxiques :

$$g(x) +_{AC} h(y) \to \ldots \text{ est remplacée par } \left\{ \begin{array}{l} g(x) + h(y) \to \ldots \\ h(y) + g(x) \to \ldots \end{array} \right.$$

10. Le compilateur ASF+SDF n'étant pas encore distribué.
11. Cet exemple n'a pas été expérimenté avec Brute parce que celui-ci n'implante pas encore de sortes *builtins*.

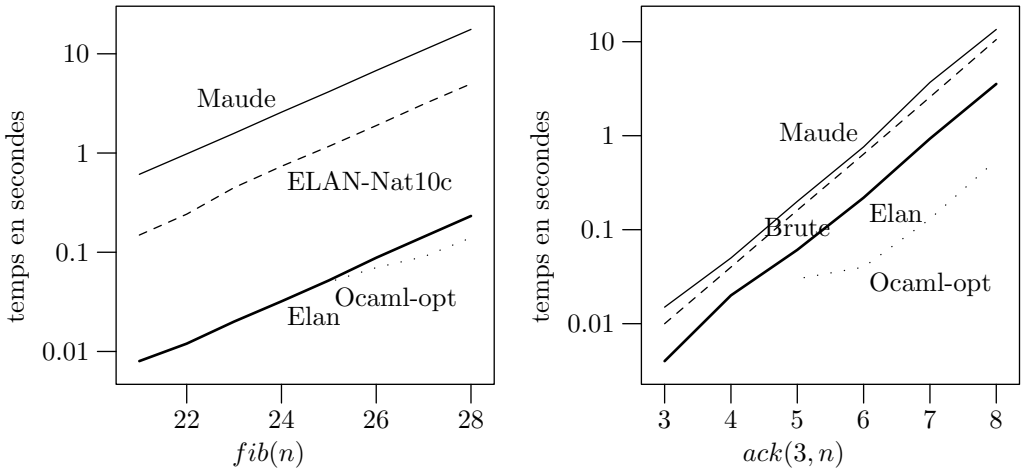


FIG. 12.10 – *Efficacité comparée sur Fib_builtin et Ackermann*

Les résultats sont intéressants parce qu'ils montrent que la spécification de l'addition et de la multiplication sur les entiers, en n'utilisant que des constructeurs et des règles de réécriture « pures » peut mener à des calculs plus efficaces que ceux réalisés en utilisant des sortes et des opérations *builtins*.

La figure de droite correspond quant à elle au calcul de la fonction d'Ackermann en utilisant des entiers représentés par des successeurs de Peano $(0,s(0),s(s(0)),\dots)$. Là encore, les techniques de compilation offrent des résultats intéressants.

À titre de comparaison, nous avons effectué ces mêmes mesures avec le compilateur Objective Caml[12]. Cette implantation du langage Caml (Cousineau et al. 1985, Weis et Leroy 1993, Cousineau et Mauny 1995, Leroy et Mauny 1993, Leroy 1995) génère du code natif optimisé qui est particulièrement efficace.

12.4.2 Calculs avec filtrage AC

C'est en utilisant ELAN pour implanter une procédure de complétion avec contraintes que j'ai commencé à découvrir la puissance et l'expressivité des symboles AC. J'ai malheureusement découvert, presque aussi vite, le prix qu'il fallait payer pour profiter de cette expressivité : une patience sans faille.

En 1996, après avoir étudié quelques problèmes liés à la réécriture modulo AC, nous avons commencé à élaborer et expérimenter de nouvelles techniques de filtrage et de normalisation modulo AC. À cette époque, Maude, Brute et le compilateur ELAN n'existaient pas encore. Claude Kirchner me montra un exemple servant de *benchmark* à l'équipe OBJ dans les années 90. Le problème Dart (voir annexe A.4) consiste à énumérer les différents scores qu'il est possible d'atteindre lorsqu'on joue aux fléchettes, sachant qu'il faut commencer par un « centre » ou un «double» pour démarrer la partie. En 1990, il fallait environ 3 heures 30 à OBJ pour résoudre un de ces problèmes de comptage. Depuis, les ordinateurs ont évolué et aujourd'hui encore, il faut plus de 12 minutes sur Sun Ultra 2 pour résoudre ce problème. Ce *benchmark* nous a longtemps permis d'expérimenter et de valider les techniques de compilation développées dans cette thèse. Les premiers résultats expérimentaux étaient encourageant et nous ont poussés à continuer nos

12. Disponible à l'adresse : http://pauillac.inria.fr/caml

efforts : il faut actuellement moins d'une seconde au compilateur ELAN pour résoudre ce même problème.

Le tableau ci-dessous permet de situer le niveau de performance des nouveaux « moteurs » que sont Brute, Maude et ELAN par rapport aux autres outils permettant d'effectuer de la normalisation modulo AC. Lorsqu'un «-» apparaît dans une case, cela signifie que l'exemple n'a pas été expérimenté sur le logiciel correspondant. Un «?» indique que le calcul a été interrompu par manque de mémoire ou de patience, ou que l'information n'est pas disponible.

	Bool3 ($n = 6$)		Nat10 ($n = 16$)		Somme ($n = 100$)	
	rwr	sec	rwr	sec	rwr	sec
CiME	?	> 24h	?	294	-	-
OBJ	?	> 24h	26,936	111	?	> 24h
OTTER	?	> 600 [13]	-	-	-	-
ReDuX	268,658	1200	-	-	-	-
RRL	?	> 4h [13]	-	-	-	-
Spike	?	> 24h	-	-	?	> 24h
Brute	34,407	2.25	26,648	0.360	177,595	6.247
Maude	4,854	0.153	25,314	0.170	177,252	16.774
ELAN	5,282	0.332	15,384	0.163	177,152	1.326

Ce tableau permet bien évidemment de savoir qui d'ELAN, de Maude ou de Brute met un dixième de seconde de plus ou de moins que l'autre, mais son principal intérêt est surtout de montrer que les exemples testés sont « difficiles » : il y a deux ans à peine, il était quasiment impossible de résoudre ces problèmes. Ce tableau permet non seulement de noter les progrès effectués en terme de performance (sec), mais il montre aussi que le nombre total de règles appliquées (rwr) est globalement en baisse.

Les figures suivantes permettent de mieux suivre les différences de performance entre Brute, ELAN et Maude.

La figure 12.11 montre que pour l'exemple Nat10, les performances d'ELAN et de Maude sont similaires. Comme nous l'avons vu précédemment, une grande partie du temps de calcul est passé à construire et détruire des structures de graphes bipartis compacts et nous pensons pouvoir réduire grandement ce coût en utilisant des techniques de « recyclage de structures ».

Les résultats de la figure 12.12 semblent donner un léger avantage à Maude, ce qui nous amène à nous demander comment et pourquoi un interpréteur irait-il plus vite qu'un compilateur ?

Il y a dans un premier temps l'effet « *benchmark* » : Maude et le compilateur ELAN ont été développés en parallèle en voyant alternativement l'un améliorer ses performances par rapport à l'autre sur ce type d'exemple. Steven Eker a par ailleurs développé d'excellentes techniques de gestion mémoire, de gestion du partage des termes et de « *greedy matching* ». Ces dernières techniques, à rapprocher de l'algorithme « glouton » présenté chapitre 6, sont des spécialisations de l'algorithme de filtrage AC qui permettent d'extraire efficacement une solution d'un problème donné. Pour cela, des heuristiques sont appliqués, mais il se peut qu'une solution existe sans qu'elle puisse être trouvée par cette classe d'algorithmes. Dans ce cas, l'algorithme de filtrage général est utilisé. Il faut aussi noter que ce type de méthode ne fonctionne que pour des motifs relativement simples et des règles de réécriture non conditionnelles.

13. Plus de 70 Mo and 115 Mo étaient respectivement utilisés avant l'arrêt du calcul.

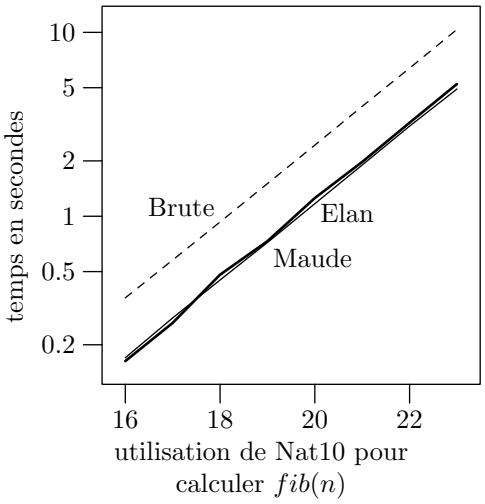


FIG. 12.11 – *Efficacité comparée sur Nat10*

La deuxième raison pour laquelle notre compilateur ne va pas forcément plus vite qu'un interpréteur peut s'expliquer par la nature des problèmes traités et le fait que de nombreuses structures de données doivent être créées dynamiquement. La résolution des graphes bipartis, par exemple, se fait de manière identique dans Brute, Maude et ELAN, même si on pourrait imaginer pré-calculer des générateurs de solutions, sachant que certains sommets du graphe (ceux qui correspondent aux motifs) sont connus à l'avance.

Le dernier exemple (figure 12.13) correspond au programme Somme présenté précédemment, il est ici utilisé pour calculer la somme des entiers de 1 à n : $(\Sigma_{i=1}^{n} i)$.

Notons que le système, défini page 177, teste effectivement la vitesse d'extraction des filtres AC, mais seulement si les règles sont appliquées avec priorité : cela assure que la première règle $\mathsf{state}(\dots)$ est essayée (sans succès) avant la seconde. Lorsque cette contrainte n'est pas assurée par le logiciel testé, et c'est le cas de Brute par exemple, il faut alors ajouter un opérateur auxiliaire et remplacer les deux dernières règles par les trois suivantes pour simuler un comportement identique :

$$
\begin{array}{lcl}
\mathsf{fire}(\mathsf{state}(s_1,s_2,y)) & \to & \mathsf{state}'(s_1,s_2,y) \\
\mathsf{state}(s_1 \cup \mathsf{set}(x),s_2,y) & \to & \mathsf{error}\ \textbf{if}\ x \in s_2 \\
\mathsf{state}'(s_1 \cup \mathsf{set}(x),s_2,y) & \to & \mathsf{fire}(\mathsf{state}(s_1,s_2 \cup \mathsf{set}(x),x+y))\ \textbf{if}\ x \notin s_2
\end{array}
$$

Le terme à réduire ($n = 100$) devenant alors :

$$\mathsf{fire}(\mathsf{state}(\emptyset \cup \mathsf{set}(1) \cup \cdots \cup \mathsf{set}(100),\emptyset,0))$$

Le chapitre 6 présentait l'utilisation des structures de graphes bipartis compacts comme un moyen d'accélérer le traitement des règles conditionnelles, et c'est effectivement ce que semblent indiquer les résultats de la figure 12.13.

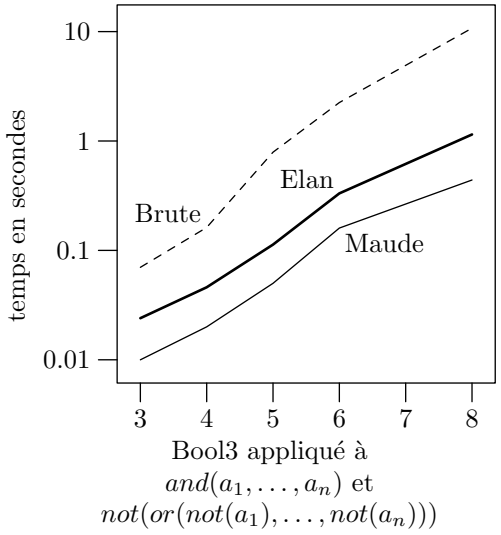


FIG. 12.12 – *Efficacité comparée sur Bool3*

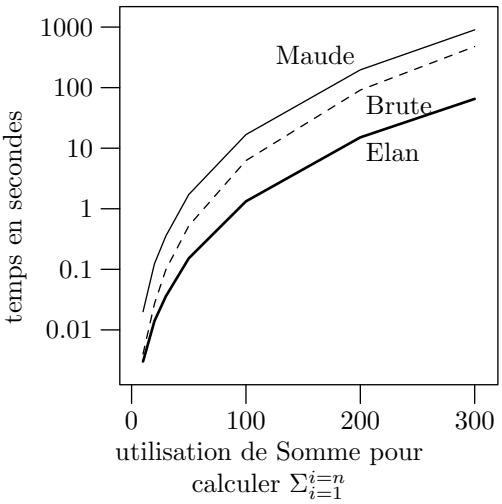


FIG. 12.13 – *Efficacité comparée sur Somme*

Conclusion

Nous voici donc arrivés au terme de cette thèse dont le fil conducteur fut la conception et la réalisation du compilateur ELAN : un langage à base de règles de réécriture et de stratégies non-déterministes. Se fondant sur la logique de réécriture présentée dans (Meseguer 1992) et permettant la définition d'opérateurs infixes, de règles conditionnelles, de symboles associatifs-commutatifs et de stratégies d'exploration non-déterministes, ELAN fait partie des langages de spécification expressifs, ayant des bases théoriques solides et concrètement utilisables pour prototyper et réaliser des applications de grande envergure. Les travaux sur ce langage participent pleinement à l'emergence de nouveaux paradigmes de programmation qui tendent à offrir une grande expressivité et qui séparent clairement le *traitement* des données du *contrôle* de ces traitements.

Le réel défi de cette thèse fut de montrer qu'un tel langage peut rester un sujet de recherche, un terrain d'expérimentation, une source d'idées nouvelles, sans pour autant être condamné à rester isolé sur une machine d'un centre de recherche.

En présence de problèmes difficiles tels que le filtrage AC et la gestion du non-déterminisme, nous avons toujours tenté de développer des solutions théoriques innovantes et d'en dériver des algorithmes qui intégrent dès leur conception les contraintes permettant d'aboutir à une implantation efficace.

Apports

Il y a des travaux qui intriguent, passionnent et finissent par s'inscrire dans les mémoires à tout jamais, et d'autres qui participent cependant à la construction d'un édifice de grande ampleur en proposant des solutions innovantes ou en caractérisant des voies infructueuses.

Bien qu'appartenant à la deuxième catégorie, les apports de cette thèse sont multiples. Au sein de l'équipe Prothéo, je pense avoir participé au travail de fond, souvent long, méticuleux et passé sous silence, qu'est la mise en place d'une plateforme de développement. À savoir, une réflexion sur l'organisation des sources du logiciel développé, l'utilisation d'un gestionnaire de versions tel que `CVS`, la réalisation de nombreux exemples et surtout la mise en place d'une procédure de test permettant de vérifier que les développements d'ELAN sont bien conservatifs : la version $n+1$ du logiciel doit être compatible avec tous les programmes qui fonctionnaient avec la version n. Au cours de nombreuses discussions avec Dominique Colnet, auteur de GNU Eiffel, il m'a souvent dit que s'il avait le choix entre perdre les sources du compilateur Eiffel ou perdre le « jeu de tests » qu'il a construit parallèlement au développement du compilateur, il préfèrerait perdre les sources de son programme. Cette mise en place de méthodes de développement rend plus facile le travail en équipe tout en assurant un avenir au logiciel, nous aide à améliorer considérablement la qualité du logiciel produit et nous a aussi permis de diffuser l'environnement ELAN par `ftp` et sur le cédérom édité par l'INRIA. Le logiciel est actuellement diffusé dans plus de

40 unités de recherche différentes. Nous avons aussi été agréablement surpris de savoir qu'ELAN est utilisé comme support de cours sur le génie logiciel, les langages de spécification algébrique et la réécriture, dans différentes universités américaines et européennes.

D'un point de vue théorique, les apports ne sont certes pas comparables au théorème de Gödel, mais se composent de nombreuses observations, propriétés et algorithmes qui apportent des solutions aux problèmes de filtrage syntaxique, de filtrage AC, d'analyse du déterminisme et de compilation de stratégies.

Compilation de la normalisation AC

La complexité des algorithmes traitant les théories AC est telle que les outils résultants sont souvent inefficaces parce que le filtrage est exponentiel. Une étude minutieuse de la gestion du cas AC par l'interpréteur ELAN a permis de mettre en évidence les problèmes à résoudre et de faire ressortir un sous-ensemble de motifs de règles de réécriture particulièrement utilisés, qu'il est intéressant de compiler efficacement. Après une première phase d'élaboration de nouvelles techniques de compilation de la réécriture Associative et Commutative, présentée dans (Moreau et Kirchner 1997), nous avons implanté un prototype permettant de tester et de valider l'intérêt des méthodes imaginées. Notre approche consiste à compiler de manière très efficace les règles qui apparaissent le plus souvent dans les spécifications écrites par les utilisateurs et à traiter les autres règles par une technique de transformation de programmes. Le cœur de la méthode repose sur la définition d'une structure de données compacte qui permet de factoriser le travail effectué pendant les processus de résolution des problèmes de filtrage AC : au lieu de construire des structures de données pour chaque nouveau problème de filtrage, celles-ci ne sont calculées qu'une seule fois et réutilisées par différentes procédures de résolution. Cette structure de données est construite en utilisant des automates de filtrage syntaxique *many-to-one*, ce qui nous a amené à proposer un nouvel algorithme de compilation du filtrage syntaxique. Cet algorithme, présenté dans le chapitre 5, accélère et permet une construction incrémentale des automates. Son principal intérêt est d'accélérer la procédure de filtrage et de réduire la taille des automates engendrés en partageant des sous-ensembles d'états.

Les méthodes utilisées ainsi que les résultats obtenus sont présentés dans (Moreau et Kirchner 1998). Cet article, qui reprend les idées du chapitre 6, ne se limite pas au cadre logique ELAN car ces techniques peuvent être utilisées pour améliorer d'autres systèmes de déduction utilisant des symboles Associatifs et Commutatifs. Cet article a été remarqué et récompensé en recevant le « EAPLS[14] Best Paper Award » en automne 1998.

Compilation de stratégies non-déterministes

Le langage ELAN a pour particularité d'intégrer un mécanisme de déduction par réécriture et un langage de stratégies qui introduit du non-déterminisme, en permettant d'explorer un espace de recherche. C'est pourquoi les techniques de compilation développées ont un caractère hybride : d'une part elles sont à rapprocher des méthodes de compilation des langages fonctionnels, pour l'aspect filtrage et simplification de termes, d'autre part elles sont à rapprocher des techniques de compilation des langages logiques, pour l'aspect non-déterministe et gestion des retours arrières.

Afin de définir des schémas de compilation simples pour les différentes constructions du langage de stratégies, nous avons poursuivi un travail débuté par Marian Vittek, consistant à définir deux primitives originales qui permettent de gérer simplement et efficacement la pose

14. European Association for Programming Languages and Systems

de points de choix. Dans le cadre d'ELAN, l'intérêt principal de cette approche est de nous permettre de définir des schémas de compilation simples et lisibles, ce qui facilite la définition de nouveaux schémas de compilation dans le cadre d'une extension du langage de stratégies. Le deuxième intérêt est d'avoir une gestion homogène et cohérente du non-déterminisme, aussi bien pour compiler les stratégies que pour compiler le filtrage AC, qui introduit lui aussi du non-déterminisme, du fait de l'existence de plusieurs solutions à un problème de filtrage AC donné. Ces travaux sont présentées dans (Moreau 1998a), et peuvent être réutilisés par les communautés Résolution de contraintes ou Prolog par exemple.

Bien qu'attentif à la simplicité et à la lisibilité du code généré, nous nous sommes aussi concentré sur l'efficacité de celui-ci. Nous avons pour cela défini un algorithme d'analyse du déterminisme, décrit par un système d'inférence de types : étant donnée une spécification, l'algorithme permet de détecter quelles sont les parties qui conduisent à des calculs déterministes. Dans ces cas, le génération du code peut être améliorée en supprimant la pose de certains points de choix. Cette optimisation permet non seulement de réduire le temps et la mémoire nécessaires au calcul, mais dans certains cas, elle permet de rendre constante la consommation mémoire d'un calcul dont l'espace mémoire était proportionnel au nombre d'étapes de réécriture effectuées. Cet algorithme d'analyse du déterminisme, présenté dans (Kirchner et Moreau 1998), permet ainsi de mener à bien un grand nombre de calculs qui n'aboutissaient pas par manque de mémoire.

Environnement de spécification

Un travail de modélisation et de conception a aussi été fait pour repenser l'architecture de l'environnement ELAN et y intégrer le nouveau compilateur capable de gérer les symboles Associatifs et Commutatifs. Ce nouveau compilateur se veut indépendant de l'interpréteur afin de pouvoir être utilisé par d'autres environnements tels que ASF+SDF, développé au CWI à Amsterdam. Cette volonté d'ouvrir notre système a donné lieu à un échange entre les deux instituts de recherche : j'ai été invité un mois par l'équipe de Paul Klint afin de mettre en place un format d'échange et des outils permettant d'intégrer notre compilateur dans leur environnement de prototypage. La conception du format d'échange à donné lieu à la rédaction d'un article présenté dans (Borovanský et al. 1998).

D'un point de vue pratique, les apports de cette thèse regroupent principalement le développement de techniques d'implantation et les nouvelles possibilités offertes par l'existence du compilateur.

Implantation

Parallèlement à l'étude et à la conception de nouveaux algorithmes de compilation, j'ai été amené à implanter toutes les méthodes proposées pour expérimenter et montrer leur intérêt pratique. La difficulté d'une telle réalisation logicielle réside non seulement dans la diversité et la complexité des algorithmes à implanter, mais aussi dans leur intégration et coopération.

Le compilateur est écrit en Java, il lit une spécification ELAN et génère un programme écrit en C, qui est lui-même jumelé à une bibliothèque de gestion du non-déterminisme écrite en assembleur. Cette multitude de paradigmes de programmation rend difficile, mais intéressant, le passage de l'un à l'autre : il faut par exemple décrire et utiliser des structures de données Java pour générer des automates de filtrage syntaxiques, ces automates étant implantés et utilisant des structures de données du langage C.

Comme mentionné précédemment, la principale difficulté d'une telle réalisation est relative à l'intégration des solutions imaginées : il ne suffit pas d'avoir une procédure de filtrage AC effi-

cace pour proposer une méthode de normalisation performante : il faut avoir une vue d'ensemble et faire en sorte que l'algorithme de filtrage permette de construire efficacement des substitutions qui seront utilisées ensuite pour calculer et construire le terme réduit correspondant à l'application d'une règle de réécriture par exemple.

Notre travail d'implantation a principalement consisté à réaliser le compilateur (15.000 lignes de Java) et la bibliothèque de support d'exécution (8.000 lignes de C). Ce travail d'implantation m'a donné des compétences particulières sur des domaines tels que la gestion de la mémoire, les mécanismes de gestion des retours arrières, la représentation des termes dans un système de calcul symbolique et l'implantation de structures compactes et efficaces par exemple. La description des algorithmes utilisés et la diffusion des sources du logiciel font que toutes ces compétences peuvent évidemment être réutilisées dans le cadre d'autres implantations.

Applications

La réalisation d'un logiciel d'une telle ampleur est rarement exempte d'erreurs. Nous estimons cependant que la version actuelle du compilateur est « stable » dans la mesure où toutes les spécifications ELAN connues à ce jour peuvent être compilées correctement. Ce qui représente plusieurs milliers de modules ELAN et plusieurs centaines de milliers de lignes générées par le compilateur.

La fiabilité et les performances du compilateur ont en particulier servi à Carlos Castro et à Thomas Genet pour expérimenter les travaux développés dans le cadre de leur thèse. L'environnement de résolution de contraintes Colette, élaboré par Carlos Castro, a permis de résoudre des problèmes de *jobshop* de taille 10×10 par exemple. L'environnement de preuve de terminaison, de complétude et d'atteignabilité, développé par Thomas Genet, est actuellement utilisée dans le cadre d'un PostDoc au CNET pour prouver la correction de protocoles de télépaiement par exemple. Dans les deux cas, la taille du code généré dépasse la centaine de milliers de lignes de C, et dans un cas comme dans l'autre, la présence du compilateur est essentielle dans la mesure où certaines preuves nécessitent plusieurs jours de calcul et impliquent l'application de plusieurs milliards de règles de réécriture. Il n'était pas envisageable de résoudre ce type de problème, dans en un temps raisonnable[15] avant l'existence d'un tel compilateur.

Perspectives

L'écriture de cette thèse semble s'achever, mais ce n'est sûrement pas le cas des travaux de recherche initiés au cours de ces dernières années.

Poursuivre l'étude des environnements de spécification, en s'intéressant particulièrement à leur architecture ainsi qu'aux moyens de coordination, devrait permettre à terme de définir des environnements ouverts capables d'intégrer et de coordonner plus facilement des outils de preuve et de résolution hétérogènes par exemple.

Continuer l'étude des langages de spécification à base de règles et de stratégies en s'intéressant particulièrement au formalisme, aux preuves et aux techniques d'implantation, devrait nous amener à définir des langages plus expressifs, plus puissants et plus efficaces.

Architecture

Les environnements de spécification sont souvent composés de modules qui communiquent en utilisant un format d'échange interne ou *ad hoc*. Nous avons présenté dans le chapitre 10 un début

15. Moins de six mois de calcul par exemple.

de réflexion sur la définition d'un format générique et d'un environnement de coordination, fondés sur la notion de termes annotés (Deursen et al. 1996, van den Brand, de Jong et Olivier 1998), non seulement pour représenter les grammaires, les termes, les règles, les stratégies, les programmes, les constructions du préprocesseur, *etc.*, mais aussi pour mettre en relation les composants de l'environnement de spécification.

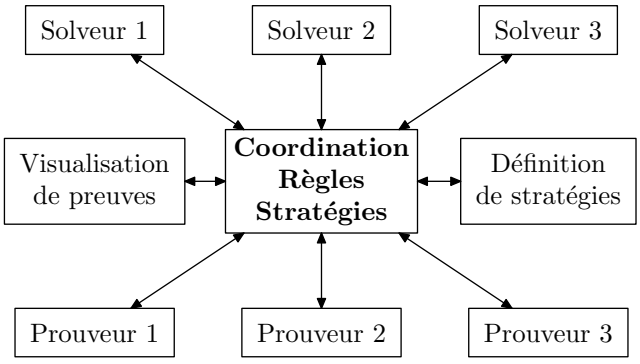
Étant donnée l'emergence récente de langages similaires à ELAN, tels que ASF+SDF, CafeOBJ, Maude, ou encore CASL développé dans le cadre Working Group ESPRIT CoFI (*Common Framework Initiative for Algebraic Specification and Development*), nous pensons que la définition d'un format d'échange « universel » permettrait de mettre en relation les différents outils développés par la communauté réécriture. Ce travail est à rapprocher des études effectuées par la communauté calcul formel, pour définir le format OpenMath (Dalmas et al. 1997), essentiellement utilisé pour représenter les problèmes à résoudre et leurs solutions.

Dans le cadre d'un environnement dont le mécanisme d'exécution repose sur l'application de règles et des stratégies, la définition d'un format à base de termes est essentielle : cela rend homogène les programmes et les données calculées par ces programmes. Ce qui permet de spécifier des outils de transformation de programmes (par évaluation partielle par exemple) ou des procédures de vérification de programmes, et de les intégrer naturellement dans l'environnement de spécification, pour transformer, optimiser ou vérifier les programmes eux-mêmes.

Disposer d'une telle architecture permettrait d'intégrer et expérimenter de nouveaux modules, de rénover certains composants, mais aussi de s'ouvrir aux autres projets en proposant des outils ayant une interface uniforme. Ce travail d'ouverture et de diffusion pourrait se faire dans le cadre du sous-groupe « Tools Task Group » du projet CoFI, dont l'objectif est de mettre des outils à disposition de la communauté CoFI. Le compilateur ELAN serait une de nos contributions au projet CoFI et serait utilisé pour compiler le sous-ensemble du langage CASL qui utilise des règles de réécriture et des stratégies par exemple.

Coopération d'outils de preuve et de résolution

Une autre application naturelle des deux thèmes de recherche mentionnés précédemment pourrait être la définition d'un langage intégrant contraintes, règles et stratégies, et utilisant des stratégies pour coordonner la coopération de démonstrateurs indépendants. À l'image de la figure suivante, l'idée consiste à utiliser toute la puissance et la souplesse d'un langage à base de règles et de stratégies pour définir la coopération entre les outils.



Dans l'optique de faciliter l'intégration et la réutilisation d'outils existants, nous envisageons d'utiliser à nouveau le format d'échange pour faire communiquer les outils entre eux. Reste à étudier précisément quelles sont les primitives nécessaires à ELAN pour qu'il puisse devenir à son

tour un outil de coordination. Le langage de stratégies servirait alors à contrôler la coopération entre différents prouveurs et solveurs, tout en gérant leur exécution parallèle ou concurrente. Nous pouvons alors imaginer des méta-stratégies chargées de distribuer les calculs en fonctions de la charge des unités de calcul disponibles. L'intérêt d'utiliser la réécriture comme langage de coopération, est qu'on dispose de méthodes et d'outils qui aident à vérifier des propriétés telles que la terminaison ou l'absence d'interbloquage par exemple.

Formalisme

En pratique, ELAN est un langage et un environnement agréable à utiliser, mais nous pensons néanmoins que l'étude du langage de stratégie, de la notion de terme de preuve et des techniques d'implantation sont des domaines prometteurs qui permettraient d'améliorer encore le formalisme et son implantation.

Intégration de *builtins* pour la résolution de contraintes. Bien qu'il soit toujours possible de spécifier un type de données en utilisant des constructeurs, des règles et des stratégies, le langage ELAN possède aussi des sortes et des opérateurs dits *élémentaires* ou *builtins*. Ceci pour des raisons évidentes d'efficacité. Une idée pourrait être d'étudier et définir une méthode systématique d'intégration de nouvelles sortes ou de nouveaux opérateurs élémentaires dans le langage. Supposons qu'on veuille effectuer des calculs intensifs utilisant des « grands » nombres par exemple. Ces travaux permettraient de définir facilement une nouvelle sorte *bignum* et d'utiliser des bibliothèques telles que `BigNum`, `GNU MP` ou `Pari` respectivement développées par l'INRIA, GNU et l'université de Bordeaux, pour implanter les opérations sur les grands nombres.

Cette idée d'intégration systématique de nouvelles sortes nous permettrait de proposer un langage à base de règles et de stratégies auquel s'ajouterait la puissance et la simplicité de la programmation par contrainte. On pourrait dans un premier temps définir une sorte *contrainte* et utiliser un résolveur tel qu'Ilog Solver, par exemple, pour prototyper les idées imaginées. À long terme, cela permettrait de mieux comprendre comment doit se faire l'intégration et de proposer une extension du langage ELAN dans laquelle termes, règles, stratégies et contraintes seraient parfaitement unifiés.

Étude des mécanismes de prétraitement. ELAN possède un mécanisme de prétraitement, appelé *préprocesseur*, qui utilise des règles et des stratégies pour engendrer de nouvelles spécifications. Le passage d'une spécification contenant des constructions du préprocesseur à une spécification ne contenant que des expansions de ces constructions, est malheureusement assez mal compris. Un projet pourrait être d'étudier les liens existant entre le préprocesseur et les notions de réflexivité. Nous envisageons ainsi de décrire complètement le comportement du préprocesseur en utilisant le formalisme ELAN lui-même. L'intérêt étant d'avoir un cadre unifié et de pouvoir raisonner, faire des preuves et des vérifications sur des programmes contenant des constructions non expansées.

Évaluation des performances d'un langage à base de règles. Un autre projet relatif au langage de spécification, concerne l'étude de méthodes d'évaluation des performances. De plus en plus de systèmes utilisent la réécriture comme moyen de calcul, pour effectuer des simplifications au sein d'un résolveur de contraintes ou d'un démonstrateur automatique, par exemple. Il existe aussi des langages, comparables à ELAN, qui utilisent la réécriture comme seul mécanisme d'évaluation (Maude, ASF+SDF et CafeOBJ par exemple). Actuellement, le seul critère utilisé, pour comparer les différentes implantation, est «le nombre de règles appliquées par seconde». Cette mesure n'est

malheureusement pas fiable parce que trop dépendante de la structure du système de réécriture évalué. Comme le montre le chapitre 12, les performances du système ELAN varient de 50.000 à 15.000.000 de règles appliquées par seconde : un facteur 300 sépare les meilleures performances des moins bonnes, en fonction des exemples testés. Nous pensons qu'il serait intéressant de définir une mesure pondérée par la « complexité » des règles pour construire un critère plus constant et surtout plus fiable, afin de pouvoir comparer l'influence des techniques d'implantation sur l'efficacité de systèmes obtenus. La complexité d'une règle pourrait se caractériser par exemple en fonction de la complexité du membre gauche, du membre droit, des conditions et de la stratégie appliquée. La complexité d'un terme pourrait se caractériser en fonction du nombre de variables, de leur linéarité, du nombre de constantes et de symboles AC par exemple.

Le concept de termes de preuves

D'un point de vue pratique, l'application de règles de réécriture sur un terme t permet de calculer une forme normale t', mais d'un point de vue théorique, cette dérivation est une preuve en logique de réécriture (Meseguer 1992) : les termes t et t' sont équivalents modulo une certaine relation de réécriture. Une idée pourrait être de rendre explicites de telles preuves en les représentant par des termes appelés *termes de preuve*. Leur construction serait alors effectuée en même temps que le calcul d'une dérivation.

Les termes de preuves permettent non seulement de représenter de manière formelle les traces d'exécution, mais ils pourraient aussi être utilisés pour analyser et « comprendre » comment un calcul s'est effectué, et aussi « rejouer » certaines parties de la preuve.

Nous proposons de définir une structure de terme de preuve capable de représenter toutes les informations calculées au cours d'une dérivation (position où une règle est appliquée, substitution utilisée pour appliquer la règle, stratégie d'application de la règle, *etc.*). En complément de cette structure, on peut imaginer l'élaboration de plusieurs outils tels que :

- un outil de visualisation qui permettrait de lire une preuve et surtout de se déplacer interactivement dedans pour mieux comprendre sa structure et le comportement d'une stratégie par exemple. Il faut alors voir le terme de preuve comme représentation de l'espace de recherche : les branches correspondant à des succès, mais aussi les branches qui mènent à des échecs sont représentées.

  La suite logique de ce travail consisterait à concevoir un outil d'exploration qui permettrait de se déplacer et de modifier certains paramètres de la preuve : on peut imaginer modifier la valeur d'un terme, le choix d'un filtre utilisé ou le choix d'une règle ou d'une stratégie appliquée par exemple, puis recalculer dynamiquement des nouveaux morceaux de la preuve, correspondant à ces changements de paramètres ;
- un filtre qui éliminerait toutes les branches menant à des échecs, ceci pour diminuer la taille de la preuve et pouvoir la donner à un démonstrateur, tel que Coq par exemple, pour vérifier la validité de la dérivation par exemple ;
- un outil d'analyse ou de déboguage qui aiderait à comprendre pourquoi un résultat attendu ne s'est pas produit, en repérant l'application d'une règle qui a fait disparaître un certain constructeur par exemple.

La structure de terme de preuve pourrait s'inspirer de la structure de terme annoté utilisée par l'environnement, l'avantage serait de pouvoir manipuler, transformer et échanger ces termes de preuve entre différents composants.

Extension du langage de stratégies : stratégies évolutives

Dans la plupart des formalismes à base de règles et de stratégies, tels que ceux définis dans ELAN ou Maude, le langage de stratégies permet de construire des expressions qui sont utilisées pour contrôler l'application des règles. L'idée est intéressante parce que la stratégie d'application des règles n'est plus figée par le système mais paramétrable par l'utilisateur. Il dispose pour cela d'un certain nombre de constructeurs pour spécifier de quelle façon un ensemble de règles doit être appliqué (veut-on un seul résultat correspondant à l'application d'une règle, tous les résultats correspondant à l'application d'une règle, ou tous les résultats correspondant à l'application de toutes les règles ?).

Malgré cette souplesse, lorsqu'on prototype un démonstrateur automatique, il arrive qu'un calcul se bloque ou diverge, simplement parce que la stratégie d'application ne permet pas de déduire un lemme particulier par exemple. Il serait alors intéressant d'offrir la possibilité à l'utilisateur d'intervenir en donnant d'autres stratégies de recherche. Actuellement, l'utilisateur doit modifier la spécification du démonstrateur et le ré-exécuter.

Stratégies interactives. Dans un premier temps, nous envisageons d'approfondir les travaux de Peter Borovanský (1998) et d'étendre le langage de stratégies actuel pour le rendre interactif et permettre la définition dynamique de nouvelles stratégies. Dans un second temps, nous envisageons de poursuivre les travaux sur les termes de preuve décrits précédemment, pour définir des outils d'analyse. La combinaison d'un langage de stratégies capable de définir dynamiquement de nouvelles stratégies, avec des outils d'analyse, permettrait de définir des démonstrateurs où la stratégie de recherche se modifierait en fonction des calculs effectués.

Stratégies « intelligentes ». Nous pensons qu'il serait intéressant de poursuivre les travaux, présentés dans les chapitres 7 et 8, portant sur l'implantation des langages de stratégies. Dans le cadre de sa thèse, Carlos Castro (1998) a modélisé des techniques de résolution de contraintes en utilisant des règles et des stratégies. Cette expérience a montré tout l'intérêt du langage de stratégies d'ELAN, mais nous a aussi donné des idées d'amélioration. En ELAN, les opérateurs de stratégies permettent d'explorer un espace de recherche en utilisant un mécanisme de retour arrière (*backtracking*), ce qui amène à explorer l'arbre de recherche en utilisant un parcours *leftmost-innermost*. Dans un problème de satisfaction de contraintes, des valeurs sont associées à des variables (X, Y et Z par exemple) et il arrive qu'un problème n'ait pas de solution tant qu'une certaine valeur est affectée à une variable ($X = 2$ par exemple). Lorsqu'on rencontre une telle situation, il faut remettre en cause l'affectation concernée pour « débloquer » le calcul. Dans le cadre d'une stratégie leftmost-innermost, implantée par un mécanisme de retour arrière classique, il faut au préalable continuer en vain l'exploration des sous-arbres de recherche (énumération des valeurs de Y et Z). Des techniques de « backtracking intelligent » ou de « *backjumping* » permettraient de se déplacer plus rapidement dans un arbre de recherche, et c'est ce que nous envisageons d'étudier dans le cadre d'ELAN.

Problèmes d'implantation : gestion mémoire, efficacité et expressivité du filtrage

Filtrage. Concernant l'étude des techniques d'implantation des langages à base de règles et de stratégies, il serait bon de continuer les travaux commencés dans cette thèse et d'étendre les algorithmes de compilation proposés pour les théories associatives-commutatives (AC) à des mélanges de AC avec d'autres axiomes comme l'idempotence ($f(x,x) = x$) et l'élément neutre

$(f(x,e) = x)$. Les techniques présentées dans le chapitre 6 peuvent alors être adaptées pour offrir à nouveau des algorithmes de filtrage et de normalisation efficaces.

Gestion mémoire. Une des caractéristiques des langages de programmation modernes est de simplifier les problèmes de gestion mémoire et d'offrir des solutions efficaces, en proposant l'utilisation d'un ramasse-miettes par exemple.

Nous pensons qu'il faut continuer à travailler sur les problèmes liés à la représentation des termes au cours d'un calcul de normalisation : faut-il représenter les termes par des listes ou des arbres ? faut-il éliminer tout partage ? autoriser un partage partiel ? ou encore partager les termes au maximum en utilisant des techniques de *hash-consing* ? Dans un souci d'économie, nous proposons d'étudier particulièrement un mécanisme de partage maximal de termes qui utiliserait des tables de hachages et une technique de *hash-consing* pour réduire l'espace mémoire nécessaire à un calcul par normalisation. Ce point est important dans la mesure où le compilateur serait utilisé pour implanter des applications (démonstrateurs ou composants de l'environnement par exemple) amenées à traiter des exemples de taille réelle (preuves de protocoles ou évaluation partielle par transformation de programmes par exemple) et donc de « gros [16] » termes.

L'étude de ces différentes représentations serait faite en parallèle avec une étude détaillée des différents algorithmes de gestion mémoire. Dans un souci d'efficacité, il serait possible de s'appuyer sur la structure des termes définis dans une spécification pour compiler des procédures de « ramasse-miettes » spécifiques. On peut ainsi imaginer précompiler des fonctions de marquage ou de copie de termes qui exploiteraient la signature des symboles de fonction pour éviter de faire du travail inutile, tel que le marquage d'un sous-terme de sorte *entier* par exemple, dans le cadre d'un algorithme de *mark and sweep*.

Efficacité. Comme le mentionnait aussi Marian Vittek en conclusion de sa thèse (1994), en réécriture, il est fréquent qu'un même calcul soit effectué plusieurs fois, aussi bien pour évaluer une condition que pour explorer un espace de recherche. Il est ainsi toujours d'actualité d'étudier un mécanisme permettant de réduire le nombre de normalisations identiques qui sont effectuées plusieurs fois. La première approche serait d'étudier une méthode de « tabulation » pour la réécriture : l'idée consiste à mémoriser dans une table les couples (terme, forme normale) les plus souvent calculés. Avant de calculer la forme normale t' d'un terme t, cette table, organisée à l'image d'une mémoire cache, serait utilisée pour y rechercher le couple (t,t'). Lorsque celui-ci est trouvé, le calcul de t à t' peut être évité.

Dans le cadre de l'exploration d'un arbre de recherche par exemple, on peut imaginer une approche différente, consistant à exploiter les termes de preuve parallèlement à la gestion des retours arrières. Lorsqu'un retour arrière (*backtracking* ou *backjumping*) est effectué, le terme de preuve peut être utilisé pour reconstituer certains calculs « détruits » (en évitant de les recalculer entièrement) pour accélérer l'exploration de l'espace de recherche. Cette technique, appelée *forwardjumping*, pourrait diminuer considérablement la redondance des calculs effectués tout en conservant la sémantique du calcul.

À plus long terme

Face à l'évolution extrêmement rapide des méthodes et des technologies liées à l'informatique, nous pensons bien que les solutions présentées dans ce document ne sont pas celles qui seront utilisées demain. Nous espérons cependant que les idées développées tout au long de cette

16. Il n'est pas rare de manipuler des termes dépassant la dizaine de méga-octets.

thèse contriburont, de près ou de loin, à l'amélioration des outils de conception des systèmes informatiques.

Il existe aujourd'hui une multitude de signaux forts qui nous rendent optimiste pour l'avenir : nous savons définir des langages de spécification ayant une grande expressivité ainsi que des bases théoriques solides, nous savons développer des méthodes de preuve automatique pour vérifier des propriétes telles que la correction ou la terminaison d'un programme, nous savons construire des implantations efficaces de langages de haut niveau, nous savons concevoir des environnements de spécification permettant de prototyper, vérifier, tester et exécuter. C'est sûrement en « intégrant » et en « coordonnant » ces différentes compétences que nous réussirons à batir de nouveaux environnements de production logicielle et à améliorer la qualité des futures générations de programmes.

En laissant notre imaginaire s'évader ainsi, cette thèse peut sembler présenter un travail « inachevé », mais n'est-ce pas là tout son intérêt ?

Annexe A

Programmes utilisés pour effectuer les expérimentations

A.1 Brute . . . 193
A.2 Caml . . . 204
A.3 Cime . . . 205
A.4 Elan . . . 211
A.5 Maude, Obj . . . 221
A.6 Otter . . . 229
A.7 Redux . . . 231
A.8 Rrl . . . 233

A.1 Brute

A.1.1 Ackermann

```
; sort declaration
(sort N)

; operator declarations
(op 0 () N ())
(op s (N) N (1))
(op ack (N N) N (1 2 0))

; rewrite rules
(rule ((I N)) (ack (0) I) (s I))
(rule ((I N)) (ack (s I) (0)) (ack I (s (0))))
(rule ((I N) (J N)) (ack (s I) (s J)) (ack I (ack (s I) J)))

(compile)
(stat on)
(reduce (ack (s (s (s (0)))) (s (0))))
(reduce (ack (s (s (s (0)))) (s (s (0)))))
```

```
(reduce (ack (s (s (s (0)))) (s (s (s (0))))))
(reduce (ack (s (s (s (0)))) (s (s (s (s (0)))))))
(reduce (ack (s (s (s (0)))) (s (s (s (s (s (0))))))))
(reduce (ack (s (s (s (0)))) (s (s (s (s (s (s (0)))))))))
(reduce (ack (s (s (s (0)))) (s (s (s (s (s (s (s (0))))))))))
(reduce (ack (s (s (s (0)))) (s (s (s (s (s (s (s (s (0)))))))))))
```

A.1.2 Bool3

```
; sort declaration
(sort B)

; operator declarations
(op b0 () B ())
(op b1 () B ())
(op b2 () B ())
(op a1 () B ())
(op a2 () B ())
(op a3 () B ())
(op a4 () B ())
(op a5 () B ())
(op a6 () B ())
(op a7 () B ())
(op a8 () B ())

(op p (B B) B (1 2 0) (:assoc :comm))
(op m (B B) B (1 2 0) (:assoc :comm))

(op and (B B) B (1 2 0))
(op or  (B B) B (1 2 0))
(op not (B)   B (1 0))
(op f (B)   B (1 0))

(op equal (B B) B (1 2 0))
(op succes () B ())

; rewrite rules
(rule ((X B)) (p  X (p X X)) (b0))
(rule ((X B)) (p  X (b0)) X)

(rule ((X B)) (m  X (m X X)) X)
(rule ((X B)) (m  X (b0)) (b0))
(rule ((X B)) (m  X (b1)) X)
(rule ((X B) (Y B) (Z B)) (m  (p X Y) Z) (p (m X Z) (m Y Z)))
(rule () (f (b2)) (p (b1) (b1)))

(rule ((X B) (Y B)) (and X Y)
        (p (m (m X X) (m Y Y))
```

```
           (p (m (f (b2)) (m (m X X) Y))
           (p (m (f (b2)) (m (m Y Y) X))
           (m (f (b2)) (m X Y))))))

(rule ((X B) (Y B)) (or X Y)
        (p (m (f (b2)) (m (m X X) (m Y Y)))
           (p (m (m X X) Y)
           (p (m (m Y Y) X)
           (p (m X Y)
           (p X Y))))))

(rule ((X B)) (not X) (p (m (f (b2)) X) (b1)))
(rule ((X B)) (equal X X) (succes))

; compile TRS
(compile)

; produce statstics
(stat on)
; q3
(reduce (equal
       (and (a1) (and (a2)(a3)))
        (not (or (not (a1)) (or (not (a2)) (not (a3)))))))
; q4
(reduce (equal
       (and (and (a1) (a2)) (and (a3) (a4)))
        (not (or (or (not (a1)) (not (a2))) (or (not (a3)) (not (a4)))))))
; q5
(reduce (equal
        (and (and (a1) (a2)) (and (a3) (and (a4) (a5))))
        (not (or (or (not (a1)) (not (a2))) (or (not (a3)) (or (not
        (a4)) (not (a5))))))))
; q6
(reduce (equal
       (and (and (and (a1) (a2)) (and (a3) (a4)))
            (and (a5) (a6)) )
        (not (or (or (or (not (a1)) (not (a2))) (or (not (a3)) (not (a4))))
            (or (not (a5)) (not (a6))) ))))
q8
(reduce (equal
       (and (and (and (a1) (a2)) (and (a3) (a4)))
            (and (and (a5) (a6)) (and (a7) (a8))))
        (not (or (or (or (not (a1)) (not (a2))) (or (not (a3)) (not (a4))))
            (or (or (not (a5)) (not (a6))) (or (not (a7)) (not (a8))))))))
```

A.1.3 Nat10

```
; sort declaration
```

```
(sort Nat Bool)

; operator declarations
(op true  () Bool ())
(op false () Bool ())
(op neq (Nat Nat) Bool (1 2 0))        ; becomes true if A==B, false otherwise.
(op neq-helper (Nat Nat) Bool (1 2 0)) ; helper operator
(op bool-reducer (Bool) Bool (1 0))    ; ditto
(op and (Bool Bool) Bool (1 2 0))      ; ditto

(op d () Nat ())
(op + (Nat Nat) Nat (1 2 0) (:assoc :comm))
(op * (Nat Nat) Nat (1 2 0) (:assoc :comm))
(op 0 (Nat)   Nat (1 0))
(op 1 (Nat)   Nat (1 0))
(op 2 (Nat)   Nat (1 0))
(op 3 (Nat)   Nat (1 0))
(op 4 (Nat)   Nat (1 0))
(op 5 (Nat)   Nat (1 0))
(op 6 (Nat)   Nat (1 0))
(op 7 (Nat)   Nat (1 0))
(op 8 (Nat)   Nat (1 0))
(op 9 (Nat)   Nat (1 0))

(op mult0 (Nat)   Nat (1 0))
(op mult1 (Nat)   Nat (1 0))
(op mult2 (Nat)   Nat (1 0))
(op mult3 (Nat)   Nat (1 0))
(op mult4 (Nat)   Nat (1 0))
(op mult5 (Nat)   Nat (1 0))
(op mult6 (Nat)   Nat (1 0))
(op mult7 (Nat)   Nat (1 0))
(op mult8 (Nat)   Nat (1 0))
(op mult9 (Nat)   Nat (1 0))

(op fib (Nat)   Nat (1 0))
(op prec (Nat)   Nat (1 0))

; rewrite rules
(rule ((A Nat) (B Nat)) (neq A B) (bool-reducer (neq-helper A B)))
(rule ((A Nat)) (neq-helper A A) (false))
(rule () (bool-reducer (false)) (false))
(rule ((A Nat) (B Nat)) (bool-reducer (neq-helper A B)) (true))
(rule ((x Bool)) (and (false) x) (false))
(rule ((x Bool)) (and x (false)) (false))
(rule ((x Bool)) (and (true) (true)) (true))

(rule () (0(d)) (d))
```

```
(rule ((x Nat)) (+ x (d)) x)

(rule ((x Nat) (y Nat)) (+ (0 x) (0 y)) (0 (+ (+ x y) (0 (d)))))
(rule ((x Nat) (y Nat)) (+ (0 x) (1 y)) (1 (+ (+ x y) (0 (d)))))
(rule ((x Nat) (y Nat)) (+ (0 x) (2 y)) (2 (+ (+ x y) (0 (d)))))
(rule ((x Nat) (y Nat)) (+ (0 x) (3 y)) (3 (+ (+ x y) (0 (d)))))
(rule ((x Nat) (y Nat)) (+ (0 x) (4 y)) (4 (+ (+ x y) (0 (d)))))
(rule ((x Nat) (y Nat)) (+ (0 x) (5 y)) (5 (+ (+ x y) (0 (d)))))
(rule ((x Nat) (y Nat)) (+ (0 x) (6 y)) (6 (+ (+ x y) (0 (d)))))
(rule ((x Nat) (y Nat)) (+ (0 x) (7 y)) (7 (+ (+ x y) (0 (d)))))
(rule ((x Nat) (y Nat)) (+ (0 x) (8 y)) (8 (+ (+ x y) (0 (d)))))
(rule ((x Nat) (y Nat)) (+ (0 x) (9 y)) (9 (+ (+ x y) (0 (d)))))

(rule ((x Nat) (y Nat)) (+ (1 x) (1 y)) (2 (+ (+ x y) (0 (d)))))
(rule ((x Nat) (y Nat)) (+ (1 x) (2 y)) (3 (+ (+ x y) (0 (d)))))
(rule ((x Nat) (y Nat)) (+ (1 x) (3 y)) (4 (+ (+ x y) (0 (d)))))
(rule ((x Nat) (y Nat)) (+ (1 x) (4 y)) (5 (+ (+ x y) (0 (d)))))
(rule ((x Nat) (y Nat)) (+ (1 x) (5 y)) (6 (+ (+ x y) (0 (d)))))
(rule ((x Nat) (y Nat)) (+ (1 x) (6 y)) (7 (+ (+ x y) (0 (d)))))
(rule ((x Nat) (y Nat)) (+ (1 x) (7 y)) (8 (+ (+ x y) (0 (d)))))
(rule ((x Nat) (y Nat)) (+ (1 x) (8 y)) (9 (+ (+ x y) (0 (d)))))
(rule ((x Nat) (y Nat)) (+ (1 x) (9 y)) (0 (+ (+ x y) (1 (d)))))

(rule ((x Nat) (y Nat)) (+ (2 x) (2 y)) (4 (+ (+ x y) (0 (d)))))
(rule ((x Nat) (y Nat)) (+ (2 x) (3 y)) (5 (+ (+ x y) (0 (d)))))
(rule ((x Nat) (y Nat)) (+ (2 x) (4 y)) (6 (+ (+ x y) (0 (d)))))
(rule ((x Nat) (y Nat)) (+ (2 x) (5 y)) (7 (+ (+ x y) (0 (d)))))
(rule ((x Nat) (y Nat)) (+ (2 x) (6 y)) (8 (+ (+ x y) (0 (d)))))
(rule ((x Nat) (y Nat)) (+ (2 x) (7 y)) (9 (+ (+ x y) (0 (d)))))
(rule ((x Nat) (y Nat)) (+ (2 x) (8 y)) (0 (+ (+ x y) (1 (d)))))
(rule ((x Nat) (y Nat)) (+ (2 x) (9 y)) (1 (+ (+ x y) (1 (d)))))

(rule ((x Nat) (y Nat)) (+ (3 x) (3 y)) (6 (+ (+ x y) (0 (d)))))
(rule ((x Nat) (y Nat)) (+ (3 x) (4 y)) (7 (+ (+ x y) (0 (d)))))
(rule ((x Nat) (y Nat)) (+ (3 x) (5 y)) (8 (+ (+ x y) (0 (d)))))
(rule ((x Nat) (y Nat)) (+ (3 x) (6 y)) (9 (+ (+ x y) (0 (d)))))
(rule ((x Nat) (y Nat)) (+ (3 x) (7 y)) (0 (+ (+ x y) (1 (d)))))
(rule ((x Nat) (y Nat)) (+ (3 x) (8 y)) (1 (+ (+ x y) (1 (d)))))
(rule ((x Nat) (y Nat)) (+ (3 x) (9 y)) (2 (+ (+ x y) (1 (d)))))

(rule ((x Nat) (y Nat)) (+ (4 x) (4 y)) (8 (+ (+ x y) (0 (d)))))
(rule ((x Nat) (y Nat)) (+ (4 x) (5 y)) (9 (+ (+ x y) (0 (d)))))
(rule ((x Nat) (y Nat)) (+ (4 x) (6 y)) (0 (+ (+ x y) (1 (d)))))
(rule ((x Nat) (y Nat)) (+ (4 x) (7 y)) (1 (+ (+ x y) (1 (d)))))
(rule ((x Nat) (y Nat)) (+ (4 x) (8 y)) (2 (+ (+ x y) (1 (d)))))
(rule ((x Nat) (y Nat)) (+ (4 x) (9 y)) (3 (+ (+ x y) (1 (d)))))

(rule ((x Nat) (y Nat)) (+ (5 x) (5 y)) (0 (+ (+ x y) (1 (d)))))
```

```
(rule ((x Nat) (y Nat)) (+ (5 x) (6 y)) (1 (+ (+ x y) (1 (d)))))
(rule ((x Nat) (y Nat)) (+ (5 x) (7 y)) (2 (+ (+ x y) (1 (d)))))
(rule ((x Nat) (y Nat)) (+ (5 x) (8 y)) (3 (+ (+ x y) (1 (d)))))
(rule ((x Nat) (y Nat)) (+ (5 x) (9 y)) (4 (+ (+ x y) (1 (d)))))

(rule ((x Nat) (y Nat)) (+ (6 x) (6 y)) (2 (+ (+ x y) (1 (d)))))
(rule ((x Nat) (y Nat)) (+ (6 x) (7 y)) (3 (+ (+ x y) (1 (d)))))
(rule ((x Nat) (y Nat)) (+ (6 x) (8 y)) (4 (+ (+ x y) (1 (d)))))
(rule ((x Nat) (y Nat)) (+ (6 x) (9 y)) (5 (+ (+ x y) (1 (d)))))

(rule ((x Nat) (y Nat)) (+ (7 x) (7 y)) (4 (+ (+ x y) (1 (d)))))
(rule ((x Nat) (y Nat)) (+ (7 x) (8 y)) (5 (+ (+ x y) (1 (d)))))
(rule ((x Nat) (y Nat)) (+ (7 x) (9 y)) (6 (+ (+ x y) (1 (d)))))

(rule ((x Nat) (y Nat)) (+ (8 x) (8 y)) (6 (+ (+ x y) (1 (d)))))
(rule ((x Nat) (y Nat)) (+ (8 x) (9 y)) (7 (+ (+ x y) (1 (d)))))

(rule ((x Nat) (y Nat)) (+ (9 x) (9 y)) (8 (+ (+ x y) (1 (d)))))

(rule ((x Nat)) (mult0 x) (d))
(rule ((x Nat)) (mult1 x) x)
(rule () (mult2 (d)) (d))
(rule () (mult3 (d)) (d))
(rule () (mult4 (d)) (d))
(rule () (mult5 (d)) (d))
(rule () (mult6 (d)) (d))
(rule () (mult7 (d)) (d))
(rule () (mult8 (d)) (d))
(rule () (mult9 (d)) (d))

(rule ((x Nat)) (mult2 (0 x)) (0 (+ (0 (d)) (mult2 x))))
(rule ((x Nat)) (mult2 (1 x)) (2 (+ (0 (d)) (mult2 x))))
(rule ((x Nat)) (mult2 (2 x)) (4 (+ (0 (d)) (mult2 x))))
(rule ((x Nat)) (mult2 (3 x)) (6 (+ (0 (d)) (mult2 x))))
(rule ((x Nat)) (mult2 (4 x)) (8 (+ (0 (d)) (mult2 x))))
(rule ((x Nat)) (mult2 (5 x)) (0 (+ (1 (d)) (mult2 x))))
(rule ((x Nat)) (mult2 (6 x)) (2 (+ (1 (d)) (mult2 x))))
(rule ((x Nat)) (mult2 (7 x)) (4 (+ (1 (d)) (mult2 x))))
(rule ((x Nat)) (mult2 (8 x)) (6 (+ (1 (d)) (mult2 x))))
(rule ((x Nat)) (mult2 (9 x)) (8 (+ (1 (d)) (mult2 x))))

(rule ((x Nat)) (mult3 (0 x)) (0 (+ (0 (d)) (mult3 x))))
(rule ((x Nat)) (mult3 (1 x)) (3 (+ (0 (d)) (mult3 x))))
(rule ((x Nat)) (mult3 (2 x)) (6 (+ (0 (d)) (mult3 x))))
(rule ((x Nat)) (mult3 (3 x)) (9 (+ (0 (d)) (mult3 x))))
(rule ((x Nat)) (mult3 (4 x)) (2 (+ (1 (d)) (mult3 x))))
(rule ((x Nat)) (mult3 (5 x)) (5 (+ (1 (d)) (mult3 x))))
(rule ((x Nat)) (mult3 (6 x)) (8 (+ (1 (d)) (mult3 x))))
```

```
(rule ((x Nat)) (mult3 (7 x)) (1 (+ (2 (d)) (mult3 x))))
(rule ((x Nat)) (mult3 (8 x)) (4 (+ (2 (d)) (mult3 x))))
(rule ((x Nat)) (mult3 (9 x)) (7 (+ (2 (d)) (mult3 x))))

(rule ((x Nat)) (mult4 (0 x)) (0 (+ (0 (d)) (mult4 x))))
(rule ((x Nat)) (mult4 (1 x)) (4 (+ (0 (d)) (mult4 x))))
(rule ((x Nat)) (mult4 (2 x)) (8 (+ (0 (d)) (mult4 x))))
(rule ((x Nat)) (mult4 (3 x)) (2 (+ (1 (d)) (mult4 x))))
(rule ((x Nat)) (mult4 (4 x)) (6 (+ (1 (d)) (mult4 x))))
(rule ((x Nat)) (mult4 (5 x)) (0 (+ (2 (d)) (mult4 x))))
(rule ((x Nat)) (mult4 (6 x)) (4 (+ (2 (d)) (mult4 x))))
(rule ((x Nat)) (mult4 (7 x)) (8 (+ (2 (d)) (mult4 x))))
(rule ((x Nat)) (mult4 (8 x)) (2 (+ (3 (d)) (mult4 x))))
(rule ((x Nat)) (mult4 (9 x)) (6 (+ (3 (d)) (mult4 x))))

(rule ((x Nat)) (mult5 (0 x)) (0 (+ (0 (d)) (mult5 x))))
(rule ((x Nat)) (mult5 (1 x)) (5 (+ (0 (d)) (mult5 x))))
(rule ((x Nat)) (mult5 (2 x)) (0 (+ (1 (d)) (mult5 x))))
(rule ((x Nat)) (mult5 (3 x)) (5 (+ (1 (d)) (mult5 x))))
(rule ((x Nat)) (mult5 (4 x)) (0 (+ (2 (d)) (mult5 x))))
(rule ((x Nat)) (mult5 (5 x)) (5 (+ (2 (d)) (mult5 x))))
(rule ((x Nat)) (mult5 (6 x)) (0 (+ (3 (d)) (mult5 x))))
(rule ((x Nat)) (mult5 (7 x)) (5 (+ (3 (d)) (mult5 x))))
(rule ((x Nat)) (mult5 (8 x)) (0 (+ (4 (d)) (mult5 x))))
(rule ((x Nat)) (mult5 (9 x)) (5 (+ (4 (d)) (mult5 x))))

(rule ((x Nat)) (mult6 (0 x)) (0 (+ (0 (d)) (mult6 x))))
(rule ((x Nat)) (mult6 (1 x)) (6 (+ (0 (d)) (mult6 x))))
(rule ((x Nat)) (mult6 (2 x)) (2 (+ (1 (d)) (mult6 x))))
(rule ((x Nat)) (mult6 (3 x)) (8 (+ (1 (d)) (mult6 x))))
(rule ((x Nat)) (mult6 (4 x)) (4 (+ (2 (d)) (mult6 x))))
(rule ((x Nat)) (mult6 (5 x)) (0 (+ (3 (d)) (mult6 x))))
(rule ((x Nat)) (mult6 (6 x)) (6 (+ (3 (d)) (mult6 x))))
(rule ((x Nat)) (mult6 (7 x)) (2 (+ (4 (d)) (mult6 x))))
(rule ((x Nat)) (mult6 (8 x)) (8 (+ (4 (d)) (mult6 x))))
(rule ((x Nat)) (mult6 (9 x)) (4 (+ (5 (d)) (mult6 x))))

(rule ((x Nat)) (mult7 (0 x)) (0 (+ (0 (d)) (mult7 x))))
(rule ((x Nat)) (mult7 (1 x)) (7 (+ (0 (d)) (mult7 x))))
(rule ((x Nat)) (mult7 (2 x)) (4 (+ (1 (d)) (mult7 x))))
(rule ((x Nat)) (mult7 (3 x)) (1 (+ (2 (d)) (mult7 x))))
(rule ((x Nat)) (mult7 (4 x)) (8 (+ (2 (d)) (mult7 x))))
(rule ((x Nat)) (mult7 (5 x)) (5 (+ (3 (d)) (mult7 x))))
(rule ((x Nat)) (mult7 (6 x)) (2 (+ (4 (d)) (mult7 x))))
(rule ((x Nat)) (mult7 (7 x)) (9 (+ (4 (d)) (mult7 x))))
(rule ((x Nat)) (mult7 (8 x)) (6 (+ (5 (d)) (mult7 x))))
(rule ((x Nat)) (mult7 (9 x)) (3 (+ (6 (d)) (mult7 x))))
```

```
(rule ((x Nat)) (mult8 (0 x)) (0 (+ (0 (d)) (mult8 x))))
(rule ((x Nat)) (mult8 (1 x)) (8 (+ (0 (d)) (mult8 x))))
(rule ((x Nat)) (mult8 (2 x)) (6 (+ (1 (d)) (mult8 x))))
(rule ((x Nat)) (mult8 (3 x)) (4 (+ (2 (d)) (mult8 x))))
(rule ((x Nat)) (mult8 (4 x)) (2 (+ (3 (d)) (mult8 x))))
(rule ((x Nat)) (mult8 (5 x)) (0 (+ (4 (d)) (mult8 x))))
(rule ((x Nat)) (mult8 (6 x)) (8 (+ (4 (d)) (mult8 x))))
(rule ((x Nat)) (mult8 (7 x)) (6 (+ (5 (d)) (mult8 x))))
(rule ((x Nat)) (mult8 (8 x)) (4 (+ (6 (d)) (mult8 x))))
(rule ((x Nat)) (mult8 (9 x)) (2 (+ (7 (d)) (mult8 x))))

(rule ((x Nat)) (mult9 (0 x)) (0 (+ (0 (d)) (mult9 x))))
(rule ((x Nat)) (mult9 (1 x)) (9 (+ (0 (d)) (mult9 x))))
(rule ((x Nat)) (mult9 (2 x)) (8 (+ (1 (d)) (mult9 x))))
(rule ((x Nat)) (mult9 (3 x)) (7 (+ (2 (d)) (mult9 x))))
(rule ((x Nat)) (mult9 (4 x)) (6 (+ (3 (d)) (mult9 x))))
(rule ((x Nat)) (mult9 (5 x)) (5 (+ (4 (d)) (mult9 x))))
(rule ((x Nat)) (mult9 (6 x)) (4 (+ (5 (d)) (mult9 x))))
(rule ((x Nat)) (mult9 (7 x)) (3 (+ (6 (d)) (mult9 x))))
(rule ((x Nat)) (mult9 (8 x)) (2 (+ (7 (d)) (mult9 x))))
(rule ((x Nat)) (mult9 (9 x)) (1 (+ (8 (d)) (mult9 x))))

(rule ((x Nat)) (* x (d)) (d))
(rule ((x Nat) (y Nat)) (* (0 x) y) (+ (0 (* x y)) (mult0 y)))
(rule ((x Nat) (y Nat)) (* (1 x) y) (+ (0 (* x y)) (mult1 y)))
(rule ((x Nat) (y Nat)) (* (2 x) y) (+ (0 (* x y)) (mult2 y)))
(rule ((x Nat) (y Nat)) (* (3 x) y) (+ (0 (* x y)) (mult3 y)))
(rule ((x Nat) (y Nat)) (* (4 x) y) (+ (0 (* x y)) (mult4 y)))
(rule ((x Nat) (y Nat)) (* (5 x) y) (+ (0 (* x y)) (mult5 y)))
(rule ((x Nat) (y Nat)) (* (6 x) y) (+ (0 (* x y)) (mult6 y)))
(rule ((x Nat) (y Nat)) (* (7 x) y) (+ (0 (* x y)) (mult7 y)))
(rule ((x Nat) (y Nat)) (* (8 x) y) (+ (0 (* x y)) (mult8 y)))
(rule ((x Nat) (y Nat)) (* (9 x) y) (+ (0 (* x y)) (mult9 y)))

(rule ((x Nat)) (prec (0 x)) (9 (prec x)))
(rule ((x Nat)) (prec (1 x)) (0 x))
(rule ((x Nat)) (prec (2 x)) (1 x))
(rule ((x Nat)) (prec (3 x)) (2 x))
(rule ((x Nat)) (prec (4 x)) (3 x))
(rule ((x Nat)) (prec (5 x)) (4 x))
(rule ((x Nat)) (prec (6 x)) (5 x))
(rule ((x Nat)) (prec (7 x)) (6 x))
(rule ((x Nat)) (prec (8 x)) (7 x))
(rule ((x Nat)) (prec (9 x)) (8 x))

(rule () (fib (d)) (1 (d)))
(rule () (fib (1 (d))) (1 (d)))
(rule ((x Nat)) (fib x) (+ (fib (prec x)) (fib (prec (prec x)))))
```

```
      (((neq x (d)) (true)) ((neq x (1 (d))) (true))))

; compile TRS
(compile)

; produce statstics
(stat on)
(reduce (fib (6 (1 (d)))))
(reduce (fib (7 (1 (d)))))
(reduce (fib (8 (1 (d)))))
(reduce (fib (9 (1 (d)))))
(reduce (fib (0 (2 (d)))))
(reduce (fib (1 (2 (d)))))
(reduce (fib (2 (2 (d)))))
(reduce (fib (3 (2 (d)))))
```

A.1.4 Somme

```
; Use flattened notation for associative operators.
(option +flat)
; sort declaration
(sort Int Set State Bool Void)

; operator declarations
(op 0 () Int ())
(op s (Int) Int (1))
(op + (Int Int) Int (1 2 0) )

(op f (Int)   Int (1 0))
(op 5 () Int ())
(op 10 () Int ())
(op 15 () Int ())
(op 25 () Int ())
(op 100 () Int ())
(op 200 () Int ())
(op 300 () Int ())

(op empty () Set ())
(op set (Int) Set (1 0))
(op buildSet (Int) Set (1 0))
(op U (Set Set) Set (1 2 0) (:assoc :comm))

(op true  () Bool ())
(op false () Bool ())
(op in (Int Set) Bool (1 2 0) )
; helper operators for 'in'.
(op in-aux (Int Set) Bool (1 2 0) )     ; helper operator #1
(op true-or-false (Bool) Bool (1 0))    ; helper operator #2
```

```
(op state     (Set Set Int) State (1 2 3 0) )
(op state-aux (Set Set Int) State (1 2 3 0) )
(op error  () State ())
(op mut (State) State (1 0) )

(op void () Void ())
(op void (State) Void (1 0))

; rewrite rules

(rule ((X State)) (void X) (void))

(rule ((X Int)) (+ X (0)) X)
(rule ((X Int) (Y Int)) (+ X (s Y)) (s (+ X Y)))

(rule ()  (f (5)) (s (s (s (s (s (0)))))))
(rule ()  (f (10)) (s (s (s (s (s (f (5))))))))
(rule ()  (f (15)) (s (s (s (s (s (f (10))))))))
(rule ()  (f (25)) (+ (f (10)) (f (15))))
(rule ()  (f (100)) (+ (f (25)) (+ (f (25)) (+ (f (25)) (f (25))))))
(rule ()  (f (200)) (+ (f (100)) (f (100))))
(rule ()  (f (300)) (+ (f (200)) (f (100))))

(rule ()  (buildSet (0)) (empty))
(rule ((I Int)) (buildSet (s I)) (U (set (s I)) (buildSet I)))

(rule ((I Int)) (in I (empty)) (false)) ; this rule can be omitted.

(rule ((I Int) (S Set)) (in I S) (true-or-false (in-aux I S)))
(rule ((I Int) (J Int) (S Set)) (in-aux I (U (set J) S)) (true) ((I J)))
(rule () (true-or-false (true)) (true))
(rule ((I Int) (S Set)) (true-or-false (in-aux I S)) (false))

(rule ((I Int) (J Int) (S1 Set) (S2 Set))
       (state (U (set I) S1) S2 J) (error)
        (((in I S2) (true))))

(rule ((J Int) (S1 Set) (S2 Set))
      (mut (state S1 S2 J)) (state-aux S1 S2 J))

(rule ((I Int) (J Int) (S1 Set) (S2 Set))
       (state-aux (U (set I) S1) S2 J)
       (mut (state S1 (U (set I) S2) (+ I J)))
       (((in I S2) (false))))

; compile TRS
(compile)
```

```
; produce statstics
(stat on)
(reduce (void (mut (state (buildSet (f (10))) (empty) (0)))))
(reduce (void (mut (state (buildSet (+ (f (10)) (f (10)))) (empty) (0)))))
(reduce (void (mut (state (buildSet (+ (f (25)) (f (5)))) (empty) (0)))))
(reduce (void (mut (state (buildSet (+ (f (25)) (f (25)))) (empty) (0)))))
(reduce (void (mut (state (buildSet (f (100))) (empty) (0)))))
(reduce (void (mut (state (buildSet (f (200))) (empty) (0)))))
(reduce (void (mut (state (buildSet (f (300))) (empty) (0)))))
```

A.2 Caml

A.2.1 Ackermann

```
type unary = O | S of unary ;;

let rec ack = function
    (O,x) -> S(x)
  | (S(x),O) -> ack(x,S(O))
  | (S(x),S(y)) -> ack(x,ack(S(x),y))
;;

ack( S(S(S(O))) , S(S(S(S(S(S(S(S(O)))))))) );;
```

A.2.2 Fib_builtin

```
let rec fib = function
    0 -> 1
  | 1 -> 1
  | n -> fib(n-1)+fib(n-2)
;;

print_int (fib 28); print_newline ();;
```

A.3 Cime

A.3.1 Bool3

```
operators
% constructors
  succes : constant
  a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,a11,a12,a13,a14 : constant
  b0, b1 : constant
% operators
  +, * : AC
  not : unary
  and, or : binary
  p, m : binary
  equal : binary
% variables
  x,y,z : variable
axioms
   x + (x + x) = b0;
   b0 + x = x;
   b1 * x = x;
   x * (x * x) = x;
   b0 * x = b0;
   (x + y) * z = (x * z) + (y * z);
   not(x) = ((b1 + b1) * x) + b1 ;
   equal(x,x) = succes;
   and(x,y) = ( ((x * x) * (y * y)) +
              ( ((b1 + b1) * ((x * x) * y)) +
              ( ((b1 + b1) * ((y * y) * x)) +
              (  (b1 + b1) * (x * y)) ))) ;
   or(x,y) =    ( ((b1 + b1) * ((x * x) * (y * y))) +
                  ( ((x * x) * y) +
                  ( ((y * y) * x) +
                  (  (x * y) + (x + y) )))) ;
order
  interactive
problems
reduce equal(and(and(and(a1, a2), and(a3, a4)),and(a5,a6)) ,
            not(or(or(or(not(a1), not(a2)), or(not(a3), not(a4))),
            or(not(a5),not(a6))))) ;
end
```

A.3.2 Nat10

```
operators
% constructors
  d : constant
  0,1,2,3,4,5,6,7,8,9 : unary
% operators
```

```
  +, * : AC
  mult0,mult1,mult2,mult3,mult4 : unary
  mult5,mult6,mult7,mult8,mult9 : unary
  prec, fact, fib : unary
% variables
  x,y : variable
axioms
  0(d) = d ;

  x + d = x ;

  0(x) + 0(y) = 0(x+y) ;
  0(x) + 1(y) = 1(x+y) ;
  0(x) + 2(y) = 2(x+y) ;
  0(x) + 3(y) = 3(x+y) ;
  0(x) + 4(y) = 4(x+y) ;
  0(x) + 5(y) = 5(x+y) ;
  0(x) + 6(y) = 6(x+y) ;
  0(x) + 7(y) = 7(x+y) ;
  0(x) + 8(y) = 8(x+y) ;
  0(x) + 9(y) = 9(x+y) ;

  1(x) + 1(y) = 2(x+y) ;
  1(x) + 2(y) = 3(x+y) ;
  1(x) + 3(y) = 4(x+y) ;
  1(x) + 4(y) = 5(x+y) ;
  1(x) + 5(y) = 6(x+y) ;
  1(x) + 6(y) = 7(x+y) ;
  1(x) + 7(y) = 8(x+y) ;
  1(x) + 8(y) = 9(x+y) ;
  1(x) + 9(y) = 0(x+y+1(d)) ;

  2(x) + 2(y) = 4(x+y) ;
  2(x) + 3(y) = 5(x+y) ;
  2(x) + 4(y) = 6(x+y) ;
  2(x) + 5(y) = 7(x+y) ;
  2(x) + 6(y) = 8(x+y) ;
  2(x) + 7(y) = 9(x+y) ;
  2(x) + 8(y) = 0(x+y+1(d)) ;
  2(x) + 9(y) = 1(x+y+1(d)) ;

  3(x) + 3(y) = 6(x+y) ;
  3(x) + 4(y) = 7(x+y) ;
  3(x) + 5(y) = 8(x+y) ;
  3(x) + 6(y) = 9(x+y) ;
  3(x) + 7(y) = 0(x+y+1(d)) ;
  3(x) + 8(y) = 1(x+y+1(d)) ;
  3(x) + 9(y) = 2(x+y+1(d)) ;
```

```
4(x) + 4(y) = 8(x+y) ;
4(x) + 5(y) = 9(x+y) ;
4(x) + 6(y) = 0(x+y+1(d)) ;
4(x) + 7(y) = 1(x+y+1(d)) ;
4(x) + 8(y) = 2(x+y+1(d)) ;
4(x) + 9(y) = 3(x+y+1(d)) ;

5(x) + 5(y) = 0(x+y+1(d)) ;
5(x) + 6(y) = 1(x+y+1(d)) ;
5(x) + 7(y) = 2(x+y+1(d)) ;
5(x) + 8(y) = 3(x+y+1(d)) ;
5(x) + 9(y) = 4(x+y+1(d)) ;

6(x) + 6(y) = 2(x+y+1(d)) ;
6(x) + 7(y) = 3(x+y+1(d)) ;
6(x) + 8(y) = 4(x+y+1(d)) ;
6(x) + 9(y) = 5(x+y+1(d)) ;

7(x) + 7(y) = 4(x+y+1(d)) ;
7(x) + 8(y) = 5(x+y+1(d)) ;
7(x) + 9(y) = 6(x+y+1(d)) ;

8(x) + 8(y) = 6(x+y+1(d)) ;
8(x) + 9(y) = 7(x+y+1(d)) ;

9(x) + 9(y) = 8(x+y+1(d)) ;

mult0(x) = d ;

mult1(x) = x ;

mult2(d) = d ;
mult2(0(x)) = 0(mult2(x)) ;
mult2(1(x)) = 2(mult2(x)) ;
mult2(2(x)) = 4(mult2(x)) ;
mult2(3(x)) = 1(mult2(x)) ;
mult2(4(x)) = 8(mult2(x)) ;
mult2(5(x)) = 0(1(d)+mult2(x)) ;
mult2(6(x)) = 2(1(d)+mult2(x)) ;
mult2(7(x)) = 4(1(d)+mult2(x)) ;
mult2(8(x)) = 6(1(d)+mult2(x)) ;
mult2(9(x)) = 8(1(d)+mult2(x)) ;

mult3(d) = d ;
mult3(0(x)) = 0(mult3(x)) ;
mult3(1(x)) = 3(mult3(x)) ;
mult3(2(x)) = 6(mult3(x)) ;
```

```
mult3(3(x)) = 9(mult3(x)) ;
mult3(4(x)) = 2(1(d)+mult3(x)) ;
mult3(5(x)) = 5(1(d)+mult3(x)) ;
mult3(6(x)) = 8(1(d)+mult3(x)) ;
mult3(7(x)) = 1(2(d)+mult3(x)) ;
mult3(8(x)) = 4(2(d)+mult3(x)) ;
mult3(9(x)) = 7(2(d)+mult3(x)) ;

mult4(d) = d ;
mult4(0(x)) = 0(mult4(x)) ;
mult4(1(x)) = 4(mult4(x)) ;
mult4(2(x)) = 8(mult4(x)) ;
mult4(3(x)) = 2(1(d)+mult4(x)) ;
mult4(4(x)) = 6(1(d)+mult4(x)) ;
mult4(5(x)) = 0(2(d)+mult4(x)) ;
mult4(6(x)) = 4(2(d)+mult4(x)) ;
mult4(7(x)) = 8(2(d)+mult4(x)) ;
mult4(8(x)) = 2(3(d)+mult4(x)) ;
mult4(9(x)) = 6(3(d)+mult4(x)) ;

mult5(d) = d ;
mult5(0(x)) = 0(mult5(x)) ;
mult5(1(x)) = 5(mult5(x)) ;
mult5(2(x)) = 0(1(d)+mult5(x)) ;
mult5(3(x)) = 5(1(d)+mult5(x)) ;
mult5(4(x)) = 0(2(d)+mult5(x)) ;
mult5(5(x)) = 5(2(d)+mult5(x)) ;
mult5(6(x)) = 0(3(d)+mult5(x)) ;
mult5(7(x)) = 5(3(d)+mult5(x)) ;
mult5(8(x)) = 0(4(d)+mult5(x)) ;
mult5(9(x)) = 5(4(d)+mult5(x)) ;

mult6(d) = d ;
mult6(0(x)) = 0(mult6(x)) ;
mult6(1(x)) = 6(mult6(x)) ;
mult6(2(x)) = 2(1(d)+mult6(x)) ;
mult6(3(x)) = 8(1(d)+mult6(x)) ;
mult6(4(x)) = 4(2(d)+mult6(x)) ;
mult6(5(x)) = 0(3(d)+mult6(x)) ;
mult6(6(x)) = 6(3(d)+mult6(x)) ;
mult6(7(x)) = 2(4(d)+mult6(x)) ;
mult6(8(x)) = 8(4(d)+mult6(x)) ;
mult6(9(x)) = 4(5(d)+mult6(x)) ;

mult7(d) = d ;
mult7(0(x)) = 0(mult7(x)) ;
mult7(1(x)) = 7(mult7(x)) ;
mult7(2(x)) = 4(1(d)+mult7(x)) ;
```

```
mult7(3(x)) = 1(2(d)+mult7(x)) ;
mult7(4(x)) = 8(2(d)+mult7(x)) ;
mult7(5(x)) = 5(3(d)+mult7(x)) ;
mult7(6(x)) = 2(4(d)+mult7(x)) ;
mult7(7(x)) = 9(4(d)+mult7(x)) ;
mult7(8(x)) = 6(5(d)+mult7(x)) ;
mult7(9(x)) = 3(6(d)+mult7(x)) ;

mult8(d) = d ;
mult8(0(x)) = 0(mult8(x)) ;
mult8(1(x)) = 8(mult8(x)) ;
mult8(2(x)) = 6(1(d)+mult8(x)) ;
mult8(3(x)) = 4(2(d)+mult8(x)) ;
mult8(4(x)) = 2(3(d)+mult8(x)) ;
mult8(5(x)) = 0(4(d)+mult8(x)) ;
mult8(6(x)) = 8(4(d)+mult8(x)) ;
mult8(7(x)) = 6(5(d)+mult8(x)) ;
mult8(8(x)) = 4(6(d)+mult8(x)) ;
mult8(9(x)) = 2(7(d)+mult8(x)) ;

mult9(d) = d ;
mult9(0(x)) = 0(mult9(x)) ;
mult9(1(x)) = 9(mult9(x)) ;
mult9(2(x)) = 8(1(d)+mult9(x)) ;
mult9(3(x)) = 7(2(d)+mult9(x)) ;
mult9(4(x)) = 6(3(d)+mult9(x)) ;
mult9(5(x)) = 5(4(d)+mult9(x)) ;
mult9(6(x)) = 4(5(d)+mult9(x)) ;
mult9(7(x)) = 3(6(d)+mult9(x)) ;
mult9(8(x)) = 2(7(d)+mult9(x)) ;
mult9(9(x)) = 1(8(d)+mult9(x)) ;

x * d = d ;

0(x) * y = 0(x*y) ;
1(x) * y = 0(x*y) + y ;
2(x) * y = 0(x*y) + mult2(y) ;
3(x) * y = 0(x*y) + mult3(y) ;
4(x) * y = 0(x*y) + mult4(y) ;
5(x) * y = 0(x*y) + mult5(y) ;
6(x) * y = 0(x*y) + mult6(y) ;
7(x) * y = 0(x*y) + mult7(y) ;
8(x) * y = 0(x*y) + mult8(y) ;
9(x) * y = 0(x*y) + mult9(y) ;


fib(d) = 1(d) ;
fib(1(d)) = 1(d) ;
```

```
  fib(0(1(x))) = fib(prec(0(1(x)))) + fib(prec(prec(0(1(x))))) ;
  fib(0(2(x))) = fib(prec(0(2(x)))) + fib(prec(prec(0(2(x))))) ;
  fib(0(3(x))) = fib(prec(0(3(x)))) + fib(prec(prec(0(3(x))))) ;
  fib(0(4(x))) = fib(prec(0(4(x)))) + fib(prec(prec(0(4(x))))) ;
  fib(0(5(x))) = fib(prec(0(5(x)))) + fib(prec(prec(0(5(x))))) ;
  fib(0(6(x))) = fib(prec(0(6(x)))) + fib(prec(prec(0(6(x))))) ;
  fib(0(7(x))) = fib(prec(0(7(x)))) + fib(prec(prec(0(7(x))))) ;
  fib(0(8(x))) = fib(prec(0(8(x)))) + fib(prec(prec(0(8(x))))) ;
  fib(0(9(x))) = fib(prec(0(9(x)))) + fib(prec(prec(0(9(x))))) ;

  fib(1(1(x))) = fib(prec(1(1(x)))) + fib(prec(prec(1(1(x))))) ;
  fib(1(2(x))) = fib(prec(1(2(x)))) + fib(prec(prec(1(2(x))))) ;
  fib(1(3(x))) = fib(prec(1(3(x)))) + fib(prec(prec(1(3(x))))) ;
  fib(1(4(x))) = fib(prec(1(4(x)))) + fib(prec(prec(1(4(x))))) ;
  fib(1(5(x))) = fib(prec(1(5(x)))) + fib(prec(prec(1(5(x))))) ;
  fib(1(6(x))) = fib(prec(1(6(x)))) + fib(prec(prec(1(6(x))))) ;
  fib(1(7(x))) = fib(prec(1(7(x)))) + fib(prec(prec(1(7(x))))) ;
  fib(1(8(x))) = fib(prec(1(8(x)))) + fib(prec(prec(1(8(x))))) ;
  fib(1(9(x))) = fib(prec(1(9(x)))) + fib(prec(prec(1(9(x))))) ;

  fib(2(x)) = fib(1(x)) + fib(0(x)) ;
  fib(3(x)) = fib(2(x)) + fib(1(x)) ;
  fib(4(x)) = fib(3(x)) + fib(2(x)) ;
  fib(5(x)) = fib(4(x)) + fib(3(x)) ;
  fib(6(x)) = fib(5(x)) + fib(4(x)) ;
  fib(7(x)) = fib(6(x)) + fib(5(x)) ;
  fib(8(x)) = fib(7(x)) + fib(6(x)) ;
  fib(9(x)) = fib(8(x)) + fib(7(x)) ;

  prec(0(x)) = 9(prec(x)) ;
  prec(1(x)) = 0(x) ;
  prec(2(x)) = 1(x) ;
  prec(3(x)) = 2(x) ;
  prec(4(x)) = 3(x) ;
  prec(5(x)) = 4(x) ;
  prec(6(x)) = 5(x) ;
  prec(7(x)) = 6(x) ;
  prec(8(x)) = 7(x) ;
  prec(9(x)) = 8(x) ;

order
  interactive
problems
  reduce fib(6(1(d))) ;
end
```

A.4 Elan

A.4.1 Ackermann

```
module ack
sort Nat;
end
operators
global
  o : Nat;
  s(@)          : (Nat) Nat;
  ack(@,@)      : (Nat Nat) Nat;
end

rules for Nat
  x,y : Nat;
global
  [] ack(o,x) => s(x)                          end
  [] ack(s(x),o) => ack(x,s(o))                end
  [] ack(s(x),s(y)) => ack(x,ack(s(x),y))      end
end
end
```

A.4.2 Bool3

```
module bool3
import eq[Bool3] bool;
end
sort Bool3;
end

operators global
  b0            : Bool3;
  b1            : Bool3;
  b2            : Bool3;

  a1            : Bool3;
  a2            : Bool3;
  a3            : Bool3;
  a4            : Bool3;
  a5            : Bool3;
  a6            : Bool3;
  a7            : Bool3;
  a8            : Bool3;

  p(@,@)        : (Bool3 Bool3) Bool3 (AC);
  m(@,@)        : (Bool3 Bool3) Bool3 (AC);

  and(@,@)      : (Bool3 Bool3) Bool3;
```

```
  or(@,@)       : (Bool3 Bool3) Bool3;
  not(@)        : (Bool3) Bool3;

  start         : bool;
end

rules for Bool3
        X,Y,Z : Bool3;
        mX,mY,mZ,tZ : Bool3;
global
  [] p(X, p(X, X)) => b0 end
  [] p(b0, X) => X end
  [] m(b1, X) => X end
  [] m(X, m(X, X)) => X end
  [] m(b0, X) => b0 end
  [] m(p(X, Y), Z) => p(m(X, Z), m(Y, Z)) end
  [] b2 => p(b1, b1) end
  [] not(X) => p(m(b2, X), b1) end
  [] and(X,Y) =>   p( m(m(X,X), m(Y,Y)),
                   p( m(b2, m(m(X,X), Y)),
                   p( m(b2, m(m(Y,Y), X)),
                   m(b2, m(X, Y)) )))
  end
  [] or(X,Y)  =>   p( m(b2, m(m(X,X), m(Y,Y))),
                   p( m(m(X,X), Y),
                   p( m(m(Y,Y), X),
                   p( m(X, Y),
                   p(X, Y) ))))
  end
end

rules for bool
global
  [] start =>
     and(and(and(a1, a2), and(a3, a4)), and(and(a5, a6), and(a7, a8)))
     ==
     not(or(or(or(not(a1), not(a2)), or(not(a3), not(a4))),
         or(or(not(a5), not(a6)), or(not(a7), not(a8)))))
  end
end
end
```

A.4.3 Dart

```
module sdart
sort Int Set;
end
operators global
```

```
  o             : Int;
  s(@)          : (Int) Int;
  plus(@,@)     : (Int Int) Int;
  mult(@,@)     : (Int Int) Int;
  five          : Int;
  ten           : Int;
  fifteen       : Int;
  twentyfive    : Int;
  fifty         : Int;

  empty         : Set;
  set(@)        : (Int) Set;
  p1(@,@)       : (Set Set) Set (AC);
  @ + @         : (Set Set) Set (AC) alias p1(@,@):;
  p2(@,@)       : (Set Set) Set (AC);
  m2(@,@)       : (Set Set) Set (AC);
  singles       : Set;
  doubles       : Set;
  triples       : Set;
  all           : Set;
  finish        : Set;
end

rules for Int
        x,y : Int;
global
  [] five         => s(s(s(s(s(o))))                      end
  [] ten          => s(s(s(s(s(five)))))                  end
  [] fifteen      => s(s(s(s(s(ten)))))                   end
  [] twentyfive   => s(s(s(s(s(s(s(s(s(fifteen))))))))))  end
  [] fifty        => plus(twentyfive,twentyfive)          end

  [] plus(x,s(y)) => s(plus(x,y))       end
  [] plus(x,o)    => x                  end
  [] mult(x,o)    => o                  end
  [] mult(x,s(y)) => plus(mult(x,y),x)  end
end

rules for Set
        S,S1,S2 : Set;
        I,J     : Int;
global
  [] p1( S,empty )      => S end
  [] p1( S,S )          => S end

  [] p2( empty,S )              => S                                    end
  [] p2( set(I),set(J) )        => set( plus(I,J) )                     end
  [] p2( p1( set(I),S1 ) , S2 ) => p1( p2( set(I),S2 ) , p2( S1,S2 )) end
```

```
  [] m2( empty,S )               => S                                  end
  [] m2( set(I),set(J) )         => set( mult(I,J) )                   end
  [] m2( p1( set(I),S1 ) ,  S2 ) => p1( m2( set(I),S2 ) , m2( S1,S2))  end

  [] singles   =>
       // 1
       set( s(o) ) + set( s(s(o)) ) + set( s(s(s(o))) ) + set( s(s(s(s(o)))) )
       +set(five) +
       // 6
       set( s(five) ) + set( s(s(five)) ) + set( s(s(s(five))) ) +
       set( s(s(s(s(five)))) ) + set( ten ) +
       // 11
       set( s(ten) ) + set( s(s(ten)) ) + set( s(s(s(ten))) ) +
       set( s(s(s(s(ten)))) ) + set( fifteen ) +
       // 16
       set( s(fifteen) ) + set( s(s(fifteen)) ) + set( s(s(s(fifteen))) ) +
       set( s(s(s(s(fifteen)))) ) + set( plus(five,fifteen) )
  end
  [] doubles   => m2( singles , set(s(s(o))) ) end
  [] triples   => m2( singles , set(s(s(s(o)))) ) end
  [] all       => p1(singles,p1(doubles,p1(triples,p1(
                      set(twentyfive),p1(set(fifty),set(o)))))) end
  [] finish    => p2( p1( doubles , set(fifty) )  , p2( all , all ) ) end
end
end
```

A.4.4 Fib_builtin

```
module fib_builtin
import global builtinInt;
end
operators global
  fib(@) : (builtinInt) builtinInt ;
end
rules for builtinInt
  n : builtinInt ;
global
   [] fib(0) => 1 end
   [] fib(1) => 1 end
   [] fib(n) => fib(n - 1) + fib(n - 2) if greater_builtinInt(n,1) end
end
end
```

A.4.5 Nat10

Notons ici l'utilisation d'un module paramétré et du pré-processeur pour engendrer automatiquement les tables d'addition et de multiplication. Le programme peut ainsi fonctionner dans

n'importe quelle base.

```
module nat10[Base]
import builtinInt list[builtinInt];
end
sort Nat;
end
operators global
  d : Nat;
  {
    (@)I : (Nat) Nat;
    mult_I(@) : (Nat) Nat;
  }_I=0...Base

  @ + @ : (Nat Nat) Nat (AC);
  @ * @ : (Nat Nat) Nat (AC);
  prec(@) : (Nat) Nat;
  fact(@) : (Nat) Nat;
  fib(@) : (Nat) Nat;

  l : list[builtinInt];
end

rules for list[builtinInt]
global
 [] l => {I.}_I=0...(Base-1) nil end
end

rules for Nat
  x,y : Nat;
  r : builtinInt;
global
  [] (d)0  => d end
  [] x + d => x end

{
  FOR EACH J:builtinInt; R:builtinInt; B:builtinInt
  SUCH THAT J:=(listExtract) elem(l)
                AND R:=() ((I+J)-((I+J)%Base))/Base
                AND B:=() (I+J)%Base
                ANDIF J>=I :{
    [] (x)I + (y)J => (x+y + (d)R )B end
  }
}_I=0...(Base-1)
  [] mult_0(x) => d end
  [] mult_1(x) => x end
{
  [] mult_I(d) => d end
```

```
  FOR EACH J:builtinInt; R:builtinInt; B:builtinInt
  SUCH THAT J:=(listExtract) elem(l)
                AND R:=() ((I*J)-((I*J)%Base))/Base
                AND B:=() (I*J)%Base
                :{
    [] mult_I((x)J) => ((d)R + mult_I(x) )B end
  }
}_I=2...(Base-1)
  [] x * d => d end
{
  [] (x)I * y => (x*y)0 + mult_I(y) end
}_I=0...(Base-1)

FOR EACH B:builtinInt
SUCH THAT B:=() Base-1 :{
  [] prec((x)0) => (prec(x))B end
}

FOR EACH I:builtinInt; B:builtinInt SUCH THAT I:=(listExtract) elem(l)
        AND B:=() I-1
        ANDIF I>0 :{
  [] prec((x)I) => (x)B end
}
  [] fact(d) => (d)1 end
  [] fact(x) => x* fact(prec(x)) end
  [] fib(d) => (d)1 end
  [] fib((d)1) => (d)1 end
  [] fib(x) => fib(prec(x)) + fib(prec(prec(x))) end
end
end
```

A.4.6 Set

Cet exemple se compose de trois modules : `set`, `sequence` et `powerset`.

```
module set[X]
import global bool int sequence[X]; end
sort X set[X];  end

operators global
  emptyset_X    : set[X];
  emptyset      : set[X] alias emptyset_X:;
  mkSet(@)      : (sequence[X]) set[X];
  @ U @         : (set[X] set[X]) set[X] pri 100;
  (@ U @)       : (set[X] set[X]) set[X] alias @ U @:;
  @ I @         : (set[X] set[X]) set[X] pri 105;
  (@ I @)       : (set[X] set[X]) set[X] alias @ I @:;
  @ \ @         : (set[X] set[X]) set[X] pri 110;
```

```
  (@ \ @)       : (set[X] set[X]) set[X] alias @ \ @:;
  @ in @        : (X set[X]) bool;
  card(@)       : (set[X]) int;
end

rules for set[X]
  L, M : sequence[X];
  E, F : X;
  S, T : set[X];
  b : bool;
global
  [] S U emptyset               => S            end
  [] mkSet(L) U mkSet(M)        => mkSet(L , M) end
  [] emptyset I S               => emptyset     end
  [] mkSet(E) I S               => T
        where b := () E in S
        choose
          try where T:= () mkSet(E)     if b
          try where T:= () emptyset     if not(b)
        end
     end
  [] mkSet(E , L) I S           => (mkSet(E) I S) U (mkSet(L) I S) end
  [] emptyset \ S               => emptyset     end
  [] mkSet(E) \ S               => T
        where b := () E in S
        choose
          try where T := () emptyset    if b
          try where T := () mkSet(E)    if not(b)
        end
     end
  [] mkSet(E , L) \ S           => (mkSet(E) \ S) U (mkSet(L) \ S) end
end

rules for  bool
  L, M : sequence[X];
  E, F : X;
  S, T : set[X];

global
  [] E in emptyset      => false        end
  [] E in mkSet(L)      => E in L       end
end

rules for int
  L : sequence[X];
  E : X;
  n,n1,n2 : int;
  b : bool;
```

```
global
  [] card(emptyset)     => 0            end
  [] card(mkSet(L))     => size(L)      end
end
end

module sequence[X]
import global bool int eq[X]; end
sort int X sequence[X]; end
operators global
  @             : (X) sequence[X];
  @ , @         : (sequence[X] sequence[X]) sequence[X] (AC);
  elem(@)       : (sequence[X]) X;
  @ in @        : (X sequence[X]) bool;
  size(@)       : (sequence[X]) int;
end

rules for sequence[X]
  S    : sequence[X];
  E,E1 : sequence[X];
global
  [] E , E      => E             end
end

rules for  bool
  S : sequence[X];
  E,F : X;
global
  [] E in E     => true                end
  [] E in E , S => true                end
  [] E in F , S => false               end
end

rules for int
  S : sequence[X];
  E : X;
global
  [] size(E)    => 1            end
  [] size(E,S)  => size(S)+1    end
end
end

module powerset[X]
import global sequence[X] sequence[set[X]] set[X] set[set[X]]; end
sort set[X] set[set[X]]; end

operators global
  P(@)          : (set[X]) set[set[X]];
```

```
  augment(@,@)  : (set[set[X]] set[X]) set[set[X]];
end

rules for set[set[X]]
  S,T   : set[X];
  L     : sequence[set[X]];
  E     : X;
  EL    : sequence[X];
global
  [] augment(emptyset,T)        => emptyset     end
  [] augment(mkSet(S), T)       => mkSet(S U T) end
  [] augment(mkSet(S , L), T)   => mkSet(S U T) U augment(mkSet(L), T) end
  [] P(emptyset)                => mkSet(emptyset)      end
  [] P(mkSet(E))                => mkSet(emptyset , mkSet(E))   end
  [] P(mkSet(E , EL))           => P(mkSet(EL)) U
                                   augment(P(mkSet(EL)), mkSet(E)) end
end
end
```

A.4.7 Somme

```
module somme
import global builtinInt bool eq[term] ;
end
sort state set;
end

operators global
  go            : state;
  init(@)       : (builtinInt) state;
  error         : state;
  state(@,@,@)  : (set set builtinInt) state;
  U(@,@)        : (set set) set (AC);
  empty         : set;
  buildSet(@)   : (builtinInt) set;
  set(@)        : (builtinInt) set;
  in(@,@)       : (builtinInt set) bool;
end

rules for bool
  I,J  : builtinInt;
  S    : set;
global
  [] in(I,empty)        => false                end
  [] in(I,U(set(J),S))  => true if I==J         end
  [] in(I,S)            => false                end
end
```

```
rules for set
  I,J : builtinInt;
  S   : set;
global
  [] buildSet(0)        => empty end
  [] buildSet(I)        => U(set(I),buildSet(I-1)) end
end

rules for state
  I,J   : builtinInt;
  S1,S2 : set;
global
  [] go => state(buildSet(100),
                 empty,
                 0)
  end

  [] init(I)    => state(buildSet(I),
                 empty,
                 0)
  end

  [] state(U(set(I),S1) , S2 , J)
     =>
     error
     if in(I,S2)
  end

  [] state(U(set(I),S1) , S2 , J)
     =>
     state(S1, U(S2,set(I)), J+I)
     if not(in(I,S2))
  end
end
end
```

A.5 Maude, Obj

A.5.1 Ackermann

```
obj ACK is
  sorts Nat .
  op o : -> Nat .
  op s : Nat -> Nat .
  op ack : Nat Nat -> Nat .

vars x y : Nat .
  eq ack(o,x) = s(x) .
  eq ack(s(x),o) = ack(x,s(o)) .
  eq ack(s(x),s(y)) = ack(x,ack(s(x),y)) .
endo

red ack(s(s(s(o))), s(o)) .
red ack(s(s(s(o))), s(s(o))) .
red ack(s(s(s(o))), s(s(s(o)))) .
red ack(s(s(s(o))), s(s(s(s(o))))) .
red ack(s(s(s(o))), s(s(s(s(s(o)))))) .
red ack(s(s(s(o))), s(s(s(s(s(s(o))))))) .
red ack(s(s(s(o))), s(s(s(s(s(s(s(o)))))))) .
red ack(s(s(s(o))), s(s(s(s(s(s(s(s(o))))))))) .
```

A.5.2 Bool3

```
obj BOOL3 is sort Bool3 .
  ops b0 b1 b2 : -> Bool3 .
  op +  : Bool3 Bool3 -> Bool3 [assoc comm] .
  op *  : Bool3 Bool3 -> Bool3 [assoc comm] .
  ops a1 a2 a3 a4 a5 a6 a7 a8 : -> Bool3 .
  op and : Bool3 Bool3 -> Bool3 .
  op or : Bool3 Bool3 -> Bool3 .
  op not : Bool3 -> Bool3 .
  ops t f : -> Bool3 .
  op equal : Bool3 Bool3 -> Bool3 .
  ops q1 q2 q3 q4 q5 q6 q8 : -> Bool3 .

vars X Y Z : Bool3 .
  eq +(b0, X) = X .
  eq *(b0, X) = b0 .
  eq *(b1, X) = X .
  eq *(+(X, Y), Z) = +(*(X, Z), *(Y, Z)) .

  eq +(X, +(X, X)) = b0 .
  eq *(X, *(X, X)) = X .

  eq and(X,Y) =   +( *(*(X, X), *(Y, Y)),
```

```
                  +( *(b2, *(*(X, X), Y)),
                  +( *(b2, *(*(Y, Y), X)),
                     *(b2, *(X, Y)) ))) .
  eq or(X,Y) =    +( *(b2, *(*(X, X), *(Y, Y))),
                  +( *(*(X, X), Y),
                  +( *(*(Y, Y), X),
                  +( *(X, Y),
                     +(X, Y) )))) .
  eq not(X) = +(*(b2, X), b1) .
  eq b2 = +(b1, b1) .

  eq equal(X,X) = t .

  eq q2 = equal(
        and(a1,a2) , not(or(not(a1), not(a2)))) .

  eq q3 = equal(
        and(a1,and(a2,a3)) , not(or(not(a1), or(not(a2),not(a3))))) .

  eq q4 = equal(
        and(and(a1,a2),and(a3, a4))
        ,
        not(or(or(not(a1),not(a2)),or(not(a3),not(a4))))) .

  eq q5 = equal(
        and(and(a1,a2),and(a3, and(a4,a5)))
        ,
        not(or(or(not(a1),not(a2)),or(not(a3),or(not(a4),not(a5)))))) .

  eq q6 = equal(
        and(and(and(a1, a2), and(a3, a4)),and(a5,a6))
        ,
        not(or(or(or(not(a1), not(a2)), or(not(a3), not(a4))),
            or(not(a5),not(a6))))) .

  eq q8 = equal(
        and(and(and(a1, a2), and(a3, a4)), and(and(a5, a6), and(a7, a8)))
        ,
        not(or(or(or(not(a1), not(a2)), or(not(a3), not(a4))),
            or(or(not(a5), not(a6)), or(not(a7), not(a8)))))) .
endo
red q2 .
red q3 .
red q4 .
red q5 .
red q6 .
red q8 .
```

A.5.3 Nat10

```
obj NAT10 is
  sort Nat .
  op d  : -> Nat .
  op +  : Nat Nat -> Nat [assoc comm] .
  op *  : Nat Nat -> Nat [assoc comm] .
  op 0 : Nat -> Nat .
  op 1 : Nat -> Nat .
  op 2 : Nat -> Nat .
  op 3 : Nat -> Nat .
  op 4 : Nat -> Nat .
  op 5 : Nat -> Nat .
  op 6 : Nat -> Nat .
  op 7 : Nat -> Nat .
  op 8 : Nat -> Nat .
  op 9 : Nat -> Nat .

  op mult0 : Nat -> Nat .
  op mult1 : Nat -> Nat .
  op mult2 : Nat -> Nat .
  op mult3 : Nat -> Nat .
  op mult4 : Nat -> Nat .
  op mult5 : Nat -> Nat .
  op mult6 : Nat -> Nat .
  op mult7 : Nat -> Nat .
  op mult8 : Nat -> Nat .
  op mult9 : Nat -> Nat .

  op fib : Nat -> Nat .
  op fact : Nat -> Nat .
  op prec : Nat -> Nat .

vars x y z : Nat .

  eq 0(d) = d .
  eq +(x,d) = x .

  eq +(0(x) , 0(y)) = 0(+(+(x,y),0(d))) .
  eq +(0(x) , 1(y)) = 1(+(+(x,y),0(d))) .
  eq +(0(x) , 2(y)) = 2(+(+(x,y),0(d))) .
  eq +(0(x) , 3(y)) = 3(+(+(x,y),0(d))) .
  eq +(0(x) , 4(y)) = 4(+(+(x,y),0(d))) .
  eq +(0(x) , 5(y)) = 5(+(+(x,y),0(d))) .
  eq +(0(x) , 6(y)) = 6(+(+(x,y),0(d))) .
  eq +(0(x) , 7(y)) = 7(+(+(x,y),0(d))) .
  eq +(0(x) , 8(y)) = 8(+(+(x,y),0(d))) .
  eq +(0(x) , 9(y)) = 9(+(+(x,y),0(d))) .
```

```
eq +(1(x) , 1(y)) = 2(+(+(x,y),0(d))) .
eq +(1(x) , 2(y)) = 3(+(+(x,y),0(d))) .
eq +(1(x) , 3(y)) = 4(+(+(x,y),0(d))) .
eq +(1(x) , 4(y)) = 5(+(+(x,y),0(d))) .
eq +(1(x) , 5(y)) = 6(+(+(x,y),0(d))) .
eq +(1(x) , 6(y)) = 7(+(+(x,y),0(d))) .
eq +(1(x) , 7(y)) = 8(+(+(x,y),0(d))) .
eq +(1(x) , 8(y)) = 9(+(+(x,y),0(d))) .
eq +(1(x) , 9(y)) = 0(+(+(x,y),1(d))) .

eq +(2(x) , 2(y)) = 4(+(+(x,y),0(d))) .
eq +(2(x) , 3(y)) = 5(+(+(x,y),0(d))) .
eq +(2(x) , 4(y)) = 6(+(+(x,y),0(d))) .
eq +(2(x) , 5(y)) = 7(+(+(x,y),0(d))) .
eq +(2(x) , 6(y)) = 8(+(+(x,y),0(d))) .
eq +(2(x) , 7(y)) = 9(+(+(x,y),0(d))) .
eq +(2(x) , 8(y)) = 0(+(+(x,y),1(d))) .
eq +(2(x) , 9(y)) = 1(+(+(x,y),1(d))) .

eq +(3(x) , 3(y)) = 6(+(+(x,y),0(d))) .
eq +(3(x) , 4(y)) = 7(+(+(x,y),0(d))) .
eq +(3(x) , 5(y)) = 8(+(+(x,y),0(d))) .
eq +(3(x) , 6(y)) = 9(+(+(x,y),0(d))) .
eq +(3(x) , 7(y)) = 0(+(+(x,y),1(d))) .
eq +(3(x) , 8(y)) = 1(+(+(x,y),1(d))) .
eq +(3(x) , 9(y)) = 2(+(+(x,y),1(d))) .

eq +(4(x) , 4(y)) = 8(+(+(x,y),0(d))) .
eq +(4(x) , 5(y)) = 9(+(+(x,y),0(d))) .
eq +(4(x) , 6(y)) = 0(+(+(x,y),1(d))) .
eq +(4(x) , 7(y)) = 1(+(+(x,y),1(d))) .
eq +(4(x) , 8(y)) = 2(+(+(x,y),1(d))) .
eq +(4(x) , 9(y)) = 3(+(+(x,y),1(d))) .

eq +(5(x) , 5(y)) = 0(+(+(x,y),1(d))) .
eq +(5(x) , 6(y)) = 1(+(+(x,y),1(d))) .
eq +(5(x) , 7(y)) = 2(+(+(x,y),1(d))) .
eq +(5(x) , 8(y)) = 3(+(+(x,y),1(d))) .
eq +(5(x) , 9(y)) = 4(+(+(x,y),1(d))) .

eq +(6(x) , 6(y)) = 2(+(+(x,y),1(d))) .
eq +(6(x) , 7(y)) = 3(+(+(x,y),1(d))) .
eq +(6(x) , 8(y)) = 4(+(+(x,y),1(d))) .
eq +(6(x) , 9(y)) = 5(+(+(x,y),1(d))) .

eq +(7(x) , 7(y)) = 4(+(+(x,y),1(d))) .
eq +(7(x) , 8(y)) = 5(+(+(x,y),1(d))) .
```

```
eq +(7(x) , 9(y)) = 6(+(+(x,y),1(d))) .

eq +(8(x) , 8(y)) = 6(+(+(x,y),1(d))) .
eq +(8(x) , 9(y)) = 7(+(+(x,y),1(d))) .

eq +(9(x) , 9(y)) = 8(+(+(x,y),1(d))) .

eq mult0(x) = d .

eq mult1(x) = x .

eq mult2(d) = d .
eq mult2(0(x)) = 0(+(0(d),mult2(x))) .
eq mult2(1(x)) = 2(+(0(d),mult2(x))) .
eq mult2(2(x)) = 4(+(0(d),mult2(x))) .
eq mult2(3(x)) = 6(+(0(d),mult2(x))) .
eq mult2(4(x)) = 8(+(0(d),mult2(x))) .
eq mult2(5(x)) = 0(+(1(d),mult2(x))) .
eq mult2(6(x)) = 2(+(1(d),mult2(x))) .
eq mult2(7(x)) = 4(+(1(d),mult2(x))) .
eq mult2(8(x)) = 6(+(1(d),mult2(x))) .
eq mult2(9(x)) = 8(+(1(d),mult2(x))) .

eq mult3(d) = d .
eq mult3(0(x)) = 0(+(0(d),mult3(x))) .
eq mult3(1(x)) = 3(+(0(d),mult3(x))) .
eq mult3(2(x)) = 6(+(0(d),mult3(x))) .
eq mult3(3(x)) = 9(+(0(d),mult3(x))) .
eq mult3(4(x)) = 2(+(1(d),mult3(x))) .
eq mult3(5(x)) = 5(+(1(d),mult3(x))) .
eq mult3(6(x)) = 8(+(1(d),mult3(x))) .
eq mult3(7(x)) = 1(+(2(d),mult3(x))) .
eq mult3(8(x)) = 4(+(2(d),mult3(x))) .
eq mult3(9(x)) = 7(+(2(d),mult3(x))) .

eq mult4(d) = d .
eq mult4(0(x)) = 0(+(0(d),mult4(x))) .
eq mult4(1(x)) = 4(+(0(d),mult4(x))) .
eq mult4(2(x)) = 8(+(0(d),mult4(x))) .
eq mult4(3(x)) = 2(+(1(d),mult4(x))) .
eq mult4(4(x)) = 6(+(1(d),mult4(x))) .
eq mult4(5(x)) = 0(+(2(d),mult4(x))) .
eq mult4(6(x)) = 4(+(2(d),mult4(x))) .
eq mult4(7(x)) = 8(+(2(d),mult4(x))) .
eq mult4(8(x)) = 2(+(3(d),mult4(x))) .
eq mult4(9(x)) = 6(+(3(d),mult4(x))) .

eq mult5(d) = d .
```

```
eq mult5(0(x)) = 0(+(0(d),mult5(x))) .
eq mult5(1(x)) = 5(+(0(d),mult5(x))) .
eq mult5(2(x)) = 0(+(1(d),mult5(x))) .
eq mult5(3(x)) = 5(+(1(d),mult5(x))) .
eq mult5(4(x)) = 0(+(2(d),mult5(x))) .
eq mult5(5(x)) = 5(+(2(d),mult5(x))) .
eq mult5(6(x)) = 0(+(3(d),mult5(x))) .
eq mult5(7(x)) = 5(+(3(d),mult5(x))) .
eq mult5(8(x)) = 0(+(4(d),mult5(x))) .
eq mult5(9(x)) = 5(+(4(d),mult5(x))) .

eq mult6(d) = d .
eq mult6(0(x)) = 0(+(0(d),mult6(x))) .
eq mult6(1(x)) = 6(+(0(d),mult6(x))) .
eq mult6(2(x)) = 2(+(1(d),mult6(x))) .
eq mult6(3(x)) = 8(+(1(d),mult6(x))) .
eq mult6(4(x)) = 4(+(2(d),mult6(x))) .
eq mult6(5(x)) = 0(+(3(d),mult6(x))) .
eq mult6(6(x)) = 6(+(3(d),mult6(x))) .
eq mult6(7(x)) = 2(+(4(d),mult6(x))) .
eq mult6(8(x)) = 8(+(4(d),mult6(x))) .
eq mult6(9(x)) = 4(+(5(d),mult6(x))) .

eq mult7(d) = d .
eq mult7(0(x)) = 0(+(0(d),mult7(x))) .
eq mult7(1(x)) = 7(+(0(d),mult7(x))) .
eq mult7(2(x)) = 4(+(1(d),mult7(x))) .
eq mult7(3(x)) = 1(+(2(d),mult7(x))) .
eq mult7(4(x)) = 8(+(2(d),mult7(x))) .
eq mult7(5(x)) = 5(+(3(d),mult7(x))) .
eq mult7(6(x)) = 2(+(4(d),mult7(x))) .
eq mult7(7(x)) = 9(+(4(d),mult7(x))) .
eq mult7(8(x)) = 6(+(5(d),mult7(x))) .
eq mult7(9(x)) = 3(+(6(d),mult7(x))) .

eq mult8(d) = d .
eq mult8(0(x)) = 0(+(0(d),mult8(x))) .
eq mult8(1(x)) = 8(+(0(d),mult8(x))) .
eq mult8(2(x)) = 6(+(1(d),mult8(x))) .
eq mult8(3(x)) = 4(+(2(d),mult8(x))) .
eq mult8(4(x)) = 2(+(3(d),mult8(x))) .
eq mult8(5(x)) = 0(+(4(d),mult8(x))) .
eq mult8(6(x)) = 8(+(4(d),mult8(x))) .
eq mult8(7(x)) = 6(+(5(d),mult8(x))) .
eq mult8(8(x)) = 4(+(6(d),mult8(x))) .
eq mult8(9(x)) = 2(+(7(d),mult8(x))) .

eq mult9(d) = d .
```

```
  eq mult9(0(x)) = 0(+(0(d),mult9(x))) .
  eq mult9(1(x)) = 9(+(0(d),mult9(x))) .
  eq mult9(2(x)) = 8(+(1(d),mult9(x))) .
  eq mult9(3(x)) = 7(+(2(d),mult9(x))) .
  eq mult9(4(x)) = 6(+(3(d),mult9(x))) .
  eq mult9(5(x)) = 5(+(4(d),mult9(x))) .
  eq mult9(6(x)) = 4(+(5(d),mult9(x))) .
  eq mult9(7(x)) = 3(+(6(d),mult9(x))) .
  eq mult9(8(x)) = 2(+(7(d),mult9(x))) .
  eq mult9(9(x)) = 1(+(8(d),mult9(x))) .

  eq *(x , d) = d .

  eq *(0(x) , y) = +(0(*(x,y)) , mult0(y)) .
  eq *(1(x) , y) = +(0(*(x,y)) , mult1(y)) .
  eq *(2(x) , y) = +(0(*(x,y)) , mult2(y)) .
  eq *(3(x) , y) = +(0(*(x,y)) , mult3(y)) .
  eq *(4(x) , y) = +(0(*(x,y)) , mult4(y)) .
  eq *(5(x) , y) = +(0(*(x,y)) , mult5(y)) .
  eq *(6(x) , y) = +(0(*(x,y)) , mult6(y)) .
  eq *(7(x) , y) = +(0(*(x,y)) , mult7(y)) .
  eq *(8(x) , y) = +(0(*(x,y)) , mult8(y)) .
  eq *(9(x) , y) = +(0(*(x,y)) , mult9(y)) .

  eq prec(0(x)) = 9(prec(x)) .
  eq prec(1(x)) = 0(x) .
  eq prec(2(x)) = 1(x) .
  eq prec(3(x)) = 2(x) .
  eq prec(4(x)) = 3(x) .
  eq prec(5(x)) = 4(x) .
  eq prec(6(x)) = 5(x) .
  eq prec(7(x)) = 6(x) .
  eq prec(8(x)) = 7(x) .
  eq prec(9(x)) = 8(x) .

  eq fact(x) = if x == d then 1(d) else *(x , fact(prec(x))) fi .

  eq fib(x) = if x == d then 1(d) else
               if x == 1(d) then 1(d) else
               +( fib(prec(x)) , fib(prec(prec(x)))) fi fi .
endo
red fib( 6(1(d)) ) .
red fib( 7(1(d)) ) .
red fib( 8(1(d)) ) .
red fib( 9(1(d)) ) .
red fib( 0(2(d)) ) .
red fib( 1(2(d)) ) .
red fib( 2(2(d)) ) .
```

```
red fib( 3(2(d)) ) .
red fib( 4(2(d)) ) .
red fib( 5(2(d)) ) .
```

A.5.4 Somme

```
obj SOMME is
  protecting MACHINE-INT .
  sorts State Set .
  vars I J     : MachineInt .
  vars S S1 S2 : Set .

  op init       : MachineInt -> State .
  op error      : -> State .
  op state      : Set Set MachineInt -> State .

  op empty      : -> Set .
  op set        : MachineInt -> Set .
  op buildSet   : MachineInt -> Set .
  op U          : Set Set -> Set [assoc comm] .

  op in         : MachineInt Set -> Bool .

  eq in(I,empty)        = false .
  ceq in(I,U(set(J),S)) = true if I == J .
  eq in(I,S)            = false .

  eq buildSet(0)        = empty .
  eq buildSet(I)        = U(set(I),buildSet(I - 1)) .

  eq init(I)            = state(buildSet(I),empty,0) .

  ceq state(U(set(I),S1) , S2 , J) = error if in(I,S2) .
  ceq state(U(set(I),S1) , S2 , J) =
      state(S1, U(S2,set(I)), J + I) if in(I,S2) == false .

endo
 red init(10) .
 red init(20) .
 red init(30) .
 red init(50) .
 red init(100) .
 red init(200) .
 red init(300) .
 red init(400) .
 red init(500) .
```

A.6 Otter

A.6.1 Bool3

```
lex([b0, b1, b2,
    a1, a2, a3, a4, a5, a6, a7, a8, success,
    q1, q2, q3, q4,
    and(_,_), or(_,_), not(_), equal(_,_),
    p(_,_), m(_,_)
 ]).
set(demod_inf).
clear(demod_history).
assign(demod_limit, -1).
assign(max_given, 1).
clear(for_sub).
clear(back_sub).
assign(max_mem, 100000).      % 100 Megabytes
list(demodulators).

p(b0, x) = x .
p(x, b0) = x .
p(x, p(x, x)) = b0 .
p(p(x, x), x) = b0 .

p(x,y)=p(y,x).
p(y,p(x,z))=p(x,p(y,z)).
m(b0, x) = b0 .
m(x, b0) = b0 .
m(b1, x) = x .
m(x, b1) = x .
m(x, m(x, x)) = x .
m(m(x, x), x) = x .
m(p(x, y), z) = p(m(x, z), m(y, z)) .
m(x,y)=m(y,x).
m(y,m(x,z))=m(x,m(y,z)).
b2 = p(b1, b1) .
and(x,y) =   p( m(m(x, x), m(y, y)),
                  p( m(b2, m(m(x, x), y)),
                  p( m(b2, m(m(y, y), x)),
                     m(b2, m(x, y)) ))) .
or(x,y) =    p( m(b2, m(m(x, x), m(y, y))),
                  p( m(m(x, x), y),
                  p( m(m(y, y), x),
                  p( m(x, y),
                     p(x, y) )))) .
not(x) = p(m(b2, x), b1) .
) .
```

```
q1 = and(and(a1, a2), and(a3, a4)) .
%q2 = not(or(or(not(a1), not(a2)), or(not(a3), not(a4)))) .
equal(x,x) = success .
end_of_list.
list(sos).
q1 .
end_of_list.
```

A.7 Redux

A.7.1 Bool3

```
DATATYPE P;
SORT
        Prop;
CONST
        a1,a2,a3,a4,a5,a6,b0,b1,success : Prop;
VAR
        x,y,z : Prop;
OPERATOR
        n: Prop -> Prop;
        a: Prop, Prop -> Prop;
        o: Prop, Prop -> Prop;
        e: Prop, Prop -> Prop;
        m: Prop, Prop -> Prop;
        p: Prop, Prop -> Prop;
        b2: -> Prop;
        q1: -> Prop;
        q2: -> Prop;
        q3: -> Prop;
NOTATION
        m,p,n,a,o,e : FUNCTION;
THEORY
        m,p : AC;
AXIOM
[1] p(x, p(x, x)) == b0 ;
[2] p(b0, x) == x ;
[3] m(b1, x) == x ;
[4] m(x, m(x, x)) == x;
[5] m(b0, x) == b0 ;
[6] m(p(x, y), z) == p(m(x, z), m(y, z));

[7] b2 == p(b1, b1) ;

[8] a(x,y) ==   p( m(m(x,x), m(y,y)),p( m(b2, m(m(x,x), y)),
                p( m(b2, m(m(y,y), x)),m(b2, m(x, y)) ))) ;

[9] o(x,y) ==   p( m(b2, m(m(x,x), m(y,y))),p( m(m(x,x), y),
                p( m(m(y,y), x),p( m(x, y),p(x, y) )))) ;

[10] n(x) == p(m(b2, x), b1) ;

[11] q1 == e(a(a1, a2),n(o(n(a1), n(a2)))) ;
[12] q2 == e(
        a(a(a1, a2), a(a3, a4)) ,
        n(o(o(n(a1), n(a2)), o(n(a3), n(a4))))
```

```
        ) ;
[13] q3 == e(
        a(a(a(a1, a2), a(a3, a4)),a(a5,a6)) ,
        n(o(o(o(n(a1), n(a2)), o(n(a3), n(a4))),o(n(a5),n(a6))))
        ) ;

[14] e(x,x) == success;
END
```

A.8 Rrl

A.8.1 Bool3

```
(init)
add
p(x, p(x, x)) == b0
p(b0, x) == x
m(b1, x) == x
m(x, m(x, x)) == x
m(b0, x) == b0
m(p(x, y), z) == p(m(x, z), m(y, z))
b2 == p(b1, b1)
a(x,y) ==   p( m(m(x,x), m(y,y)),p( m(b2, m(m(x,x), y)),
            p( m(b2, m(m(y,y), x)),m(b2, m(x, y)) )))

o(x,y) ==   p( m(b2, m(m(x,x), m(y,y))),p( m(m(x,x), y),
            p( m(m(y,y),x), p( m(x, y),p(x, y) ))))
n(x) == p(m(b2, x), b1)
]
oper
ac
m
oper
ac
p
oper
pred
b2 a o n m p b1 b0

kb

opt
prove
f
opt
brake
nom
10000000

prove
 a(a(a1, a2), a(a3, a4)) == n(o(o(n(a1), n(a2)), o(n(a3), n(a4))))
y
quit
```

Bibliographie

Aho, A. V. et Corasick, M. J. (1975). Efficient string matching - an aid to bibliographic search, *Communications of the ACM* **18**(6): 333–340.

Aho, A. V., Sethi, R. et Ullman, J. D. (1989). *Compilateurs : principes, techniques et outils*, InterEdition. ISBN 2-7296-0295-X.

Aït-Kaci, H. (1990). The WAM: a (real) tutorial, *Technical report 5*, Digital Systems Research Center, Paris (France).

Apt, K. R. et Schaerf, A. (1997). Search and imperative programming, *24th POPL*, pp. 67–79.

Bachmair, L., Chen, T. et Ramakrishnan, I. V. (1993). Associative-commutative discrimination nets, *in* M.-C. Gaudel et J.-P. Jouannaud (eds), *TAPSOFT'93: Theory and Practice of Software Development, 4th International Joint Conference CAAP/FASE*, Vol. 668 of *Lecture Notes in Computer Science*, Springer-Verlag, Orsay, France, pp. 61–74.

Bailey, S. W. (1995). *Hielp, a fast interactive lazy functional language system*, PhD thesis, University of Chicago, USA.

Bartlett, J. F. (1988). Compacting garbage collection with ambiguous roots, *Technical Report WRL-TR-88.2*, Western Research Laboratory.

Battiston, E., de Cindio, F. et Mauri, G. (1988). Objsa nets: Obj2 and petri nets for specifying concurrent systems, *Technical report*, Dipartimento di Scienze dell'Informazione Milano.

Benanav, D., Kapur, D. et Narendran, P. (1987). Complexity of matching problems, *Journal of Symbolic Computation* **3**(1 & 2): 203–216.

Bergstra, J. et Klint, P. (1995). The Discrete Time ToolBus, *Technical report*, University of Amsterdam.

Boehm, H. et Weiser, M. (1988). Garbage collection in an uncooperative environment, *Software Practice and Experience* **18**: 807–820.

Borovanský, P. (1998). *Le contrôle de la réécriture: étude et implantation d'un formalisme de stratégies*, Thèse de Doctorat d'Université, Université Henri Poincaré – Nancy 1, France.

Borovanský, P. et Castro, C. (1998). Cooperation of Constraint Solvers: Using the New Process Control Facilities of ELAN, *in* C.Kirchner et H.Kirchner (eds), *Proceedings of the 2nd International Workshop on Rewriting Logic and its Applications, RWLW'98 (Pont-à-Mousson, France)*, Vol. 15, Electronic Notes in Theoretical Computer Science, pp. 379 – 398.

Borovanský, P., Jamoussi, S., Moreau, P.-E. et Ringeissen, C. (1998). Handling ELAN rewrite programs via an exchange format, *in* C. Kirchner et H. Kirchner (eds), *Proceedings of the 2nd International Workshop on Rewriting Logic and its Applications, WRLA'98*, Vol. 15, Electronic Notes in Theoretical Computer Science, Pont-à-Mousson (France).

Borovanský, P., Kirchner, C., Kirchner, H., Moreau, P.-E. et Vittek, M. (1996). ELAN: A logical framework based on computational systems, *in* J. Meseguer (ed.), *Proceedings of the 1st International Workshop on Rewriting Logic and its Applications, RWLW'96, (Asilomar,*

Pacific Grove, CA, USA), Vol. 4, Electronic Notes in Theoretical Computer Science. **URL:** *http://www.loria.fr/~borovan/bkkmv.WRLG96.ps*

Borovanský, P., Kirchner, C., Kirchner, H., Moreau, P.-E. et Vittek, M. (1997). ELAN *V 2.0 User Manual*, first edn, Inria Lorraine & Crin, Nancy (France).

Boudet, A., Contejean, E. et Devie, H. (1990). A new AC unification algorithm with a new algorithm for solving diophantine equations, *Proceedings 5th IEEE Symposium on Logic in Computer Science, Philadelphia (Pa., USA)*, pp. 289–299.

Bouhoula, A., Jouannaud, J.-P. et Meseguer, J. (1997). Specification and proof in membership equational logic, *in* M. Bidoit et M. Dauchet (eds), *Proceedings Theory and Practice of Software, TAPSOFT'97, Development, (Lille, France)*, Vol. 1214 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 67–92.

Bouhoula, A., Kounalis, E. et Rusinowitch, M. (1992). Spike: An automatic theorem prover, *Proceedings of the 1st International Conference on Logic Programming and Automated Reasoning, St. Petersburg (Russia)*, Vol. 624 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag, pp. 460–462.

Brus, T. H., van Eskelen, M. C. J. D., van Leer, M. O. et Plasmeijer, M. J. (1986). Clean. a language for functional graph rewriting, *Internal report 95*, Computing Science Department, University of Nijmegen.

Budd, T. (1982). An implementation of generators in C, *Computer Languages* **7**: 69–87.

Bündgen, R. (1993). Reduce the redex ← ReDuX, *in* C. Kirchner (ed.), *Rewriting Techniques and Applications, 5th International Conference, RTA-93*, LNCS 690, Springer-Verlag, Montreal, Canada, pp. 446–450.

Caseau, Y. et Laburthe, F. (1996). Introduction to the CLAIRE programming language, *Technical report 96-15*, LIENS Technical.

Castro, C. (1998). *Une approche déductive de la résolution de problèmes de satisfaction de contraintes*, Thèse de Doctorat d'Université, Université Henri Poincaré – Nancy 1, France.

Cavenaghi, C., de Zanet, M. et Mauri, G. (1987). Mc-obj: a c interpreter for obj, *Technical report*, Dipmentarto Scienze dell'Informazione, Universita di Milano (Italy).

Cheney, C. J. (1970). A non-recursive list compacting algorithm, *Communications of the ACM* **13**(11): 677–668.

Christian, J. (1993). Flatterms, discrimination nets, and fast term rewriting, *Journal of Automated Reasoning* **10**(1): 95–113.

Christopher, P.-G. (1988). The specification and controlled implementation of a configuration management tool using OBJ and Ada, *in* D. Coleman, R. Gallimore et J. Goguen (eds), *Experience with OBJ*, Addison-Wesley.

Clavel, M. (1998). *Reflection in general logics, rewriting logic, and Maude*, PhD thesis, University of Navarre, Spain.

Clavel, M., Durán, F., Eker, S., Lincoln, P. et Meseguer, J. (1998). An Introduction to Maude (Beta Version), *Technical report*, SRI International, Computer Science Laboratory, Menlo Park, (CA, USA). **URL:** *ftp://ftp.csl.sri.com/pub/rewriting/beta/maude-beta-doc.ps*

Clavel, M., Eker, S., Lincoln, P. et Meseguer, J. (1996). Principles of Maude, *in* J. Meseguer (ed.), *Proceedings of the first international workshop on rewriting logic*, Vol. 4, Electronic Notes in Theoretical Computer Science, Asilomar (California).

Clavel, M. et Meseguer, J. (1996). Reflection and Strategies in Rewriting Logic, *in* J. Meseguer (ed.), *Proceedings of the 1st International Workshop on Rewriting Logic and its Appli-*

cations, RWLW'96, (Asilomar, Pacific Grove, CA, USA), Vol. 5 of *Electronic Notes in Theoretical Computer Science*, North Holland.

Codognet, P. et Diaz, D. (1995). wamcc : Compiling Prolog to C, *Proceedings of International Conference on Logic Programming*, MIT Press, Tokyo, Japan.

Collavizza, H. (1989). Première évaluation du logiciel OBJ3 pour la preuve formelle des circuits digitaux, *Rapport de Recherche 89-01*, Université de Provence, Marseille.

Collavizza, H. et Pierre, L. (1988). Formal verification of hardware using OBJ and the Boyer-Moore theorem prover, *Rapport de Recherche 88-04*, Université de Provence, Marseille.

Colnet, D., Coucaud, P. et Zendra, O. (1998). Compiler Support to Customize the Mark and Sweep Algorithm, *ACM SIGPLAN International Symposium on Memory Management (ISMM'98)*, pp. 154–165.

Contejean, E., Marché, C. et Rabehasaina, L. (1997). Rewrite systems for natural, integral, and rational arithmetic, *in* H. Comon (ed.), *Proceedings of 8-th International Conference Rewriting Techniques and Applications*, Lecture Notes in Computer Science, Springer-Verlag, Sitges, Spain, pp. 98–112.

Cousineau, G. et Mauny, M. (1995). *Approche fonctionnelle de la programmation*, Ediscience. ISBN 2-84074-114-8.

Cousineau, G., Paulson, L. C., Huet, G., Milner, R., Gordon, M. et Wadsworth, C. (1985). *The ML Handbook*, INRIA, Rocquencourt.

Crelier, R. (1994). *Separate Compilation and Module Extension*, PhD thesis, Swiss Federal Institute of Technology Zurich, Swiss.

Dalmas, S., Gaëtano, M. et Sausse, A. (1996). A distributed and cooperative environment for computer algebra, *Journal of Symbolic Computation* **21**(4-6): 427–439.

Dalmas, S., Gaëtano, M. et Watt, S. (1997). An OpenMath 1.0 implementation, *in* W. W. Küchlin (ed.), *ISSAC '97. Proceedings of the 1997 International Symposium on Symbolic and Algebraic Computation, 21–23, 1997, Maui, Hawaii*, ACM Press, New York, NY 10036, USA, pp. 241–248. **URL:** *http://www.acm.org:80/pubs/citations/proceedings/issac/258726/p241-dalmas/*

Delahaye, J.-P. (1995). *Logique, informatique et paradoxes*, Pour la Science, Diffusion Belin. ISBN 2-9029-1894-1.

Demoen, B. et Maris, G. (1994). A comparison of some schemes for translating logic to C, *Workshop on Implementations of the 11th International Conference of Logic Programming*, MIT Press, Santa Margherita, Italy.

Demoen, B. et Sagonas, K. (1998). CAT: the Copying Approach to Tabling, *"Principles of Declarative Programming"*, number 1490 in *Lecture Notes in Computer Science*, Springer-Verlag, pp. 21–35.

Deursen, A., Heering, J. et Klint, P. (1996). *Language Prototyping*, World Scientific. ISBN 981-02-2732-9.

Diaconescu, R. (1996). Foundations of Behavioural Specification in Rewriting Logic, *in* J. Meseguer (ed.), *Proceedings of the 1st International Workshop on Rewriting Logic and its Applications, RWLW'96, (Asilomar, Pacific Grove, CA, USA)*, Vol. 4, Electronic Notes in Theoretical Computer Science, pp. 225–244.

Diaconescu, R. et Futatsugi, K. (1996). Logical Semantics of CafeOBJ, *Technical Report IS-RR-96-0024S*, Japan Advanced Institute of Science and Technilogy, JAIST, Ishikawa (Japan). **URL:** *http://ldl-www.jaist.ac.jp:8080/cafeobj/abstracts/Logical-Semantics-of-CafeOBJ.html*

Diaz, D. (1995). *Étude de la compilation des langages logiques de programmation par contraintes sur les domaines finis : le systeme clp(FD)*, Thèse de Doctorat d'Université, Université d'Orleans, France.

Didrich, K., Fett, A., Gerke, C., Grieskamp, W. et Pepper, P. (1994). OPAL: Design and implementation of an algebraic programming language, *in* J. Gutknecht (ed.), *Programming Languages and System Architectures PLSA'94*, Vol. 782 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 228–244.

Doligez, D. (1995). *Conception, réalisation et certification d'un glaneur de cellules concurrent*, Thèse de Doctorat d'Université, Université Paris 7, France.

Doligez, D. et Leroy, X. (1993). A concurrent, generational garbage collector for a multithreaded implementation of ml, *Proceedings of the Symposium on Principles of Programmings Languages*, ACM, ACM, pp. 113–123.

Domenjoud, E. (1991). Solving systems of linear diophantine equations: An algebraic approach, *in* A. Tarlecki (ed.), *Proceedings 16th International Symposium on Mathematical Foundations of Computer Science, Kazimierz Dolny (Poland)*, Vol. 520 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 141–150.

Earley, J. (1970). An efficient context-free parsing algorithm, *Communications of the ACM* **13**(2): 94–102.

Eker, S. (1991). Verification of a line drawing architecture using obj3*, *Technical report*, Royal Holloway and Bedford College.

Eker, S. (1995). Associative-commutative matching via bipartite graph matching, *Computer Journal* **38**(5): 381–399.

Eker, S. (1996). Fast matching in combination of regular equational theories, *in* J. Meseguer (ed.), *Proceedings of the 1st International Workshop on Rewriting Logic and its Applications, RWLW'96, (Asilomar, Pacific Grove, CA, USA)*, Vol. 4, Electronic Notes in Theoretical Computer Science.

Forgaard, R. et Guttag, J. V. (1984). Reve: A term rewriting system generator with failure-resistant Knuth-Bendix, *Technical report*, MIT-LCS.

Fukuda, K. et Matsui, T. (1989). Finding all the perfect matchings in bipartite graphs, *Technical Report B-225*, Department of Information Sciences, Tokyo Institute of Technology, Ohokayama, Meguro-ku, Tokyo 152, Japan.

Futatsugi, K. et Diaconescu, R. (1997). CafeOBJ Report, *Technical Report in preparation*, Japan Advanced Institute of Science and Technilogy, JAIST, Ishikawa (Japan).

Futatsugi, K., Goguen, J. A., Jouannaud, J.-P. et Meseguer, J. (1984). The language OBJ-2: Its syntax, semantics and implementation, *Technical report*, SRI International, Computer Science Laboratory, Menlo Park, (CA, USA).

Futatsugi, K., Goguen, J. A., Jouannaud, J.-P. et Meseguer, J. (1985). Principles of OBJ-2, *in* B. Reid (ed.), *Proceedings 12th ACM Symposium on Principles of Programming Languages*, ACM, pp. 52–66.

Futatsugi, K., Goguen, J. A., Meseguer, J. et Okada, K. (1987). Parameterized programming in OBJ-2, *in* R. Balzer (ed.), *Proceedings of Ninth International Conference on Software Engineering*, IEEE Computer Society Press, (Monterey, CA (USA)), pp. 51–60.

Futatsugi, K. et Nakagawa, A. (1996). An Overview of Cafe Project, *Proceedings of Fist CafeOBJ workshop*, Yokohama (Japan).

Futatsugi, K. et Sawada, T. (1994). Cafe as an extensible specification environment, *Proceedings of the Kunming International CASE Symposium*.

Genet, T. (1998). *Contraintes d'ordre et automates d'arbre pour les preuves de terminaison*, Thèse de Doctorat d'Université, Université Henri Poincaré – Nancy 1, France.

Goguen, J. A. (1977). Abstract errors for abstract data types, *in* E. Neuhold (ed.), *Formal Description of Programming Concepts, Amsterdam (The Nederlands)*, Elsevier Science Publishers B. V. (North-Holland).

Goguen, J. A. (1978). Some design principles and theory for OBJ-0, a language for expressing and executing algebraic specifications of programs, *in* E. Blum, M. Paul et S. Takasu (eds), *Proceedings of Mathematical Studies of Information Processing*, Vol. 75, Lecture Notes in Computer Science.

Goguen, J. A. (1988a). A brief history of OBJ, *in* D. Coleman, R. Gallimore et J. Goguen (eds), *Experience with OBJ*, Addison-Wesley.

Goguen, J. A. (1988b). OBJ as a theorem prover with application to hardware verification, *Technical Report SRI-CSL-88-4R2*, SRI.

Goguen, J. A., Kirchner, C., Kirchner, H., Mégrelis, A., Meseguer, J. et Winkler, T. (1987). An introduction to OBJ-3, *in* J.-P. Jouannaud et S. Kaplan (eds), *Proceedings 1st International Workshop on Conditional Term Rewriting Systems, Orsay (France)*, Vol. 308 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 258–263. Also as internal report CRIN: 88-R-001.

Goguen, J. A., Meseguer, J. et Plaisted, D. (1982). Programming with parameterized abstract objects in OBJ., *Theory And Practice of Software Technology* pp. 163–193.

Goguen, J. A. et Tardo, J. (1977). OBJ-0 preliminary users manual, *Semantics and Theory of Computation, Technical Report 10*, UCLA, Los Angeles (USA).

Gräf, A. (1991). Left-to-rigth tree pattern matching, *in* R. V. Book (ed.), *Proceedings 4th Conference on Rewriting Techniques and Applications, Como (Italy)*, Vol. 488 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 323–334.

Graf, P. (1996). *Term Indexing*, Vol. 1053 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag.

Guttag, J. V., Horning, J. J., Garland, S. J., Jones, K. D., Modet, A. et Wing, J. M. (1993). *Larch: Languages and Tools for Formal Specification*, Springer-Verlag.

Hamel, L. H. (1995). *Behavioural Verification and Implementation of an Optimising Compiler for OBJ3*, PhD thesis, Oxford University Computing Laboratory, GB.

Henderson, F., Conway, T. et Somogyi, Z. (1996). The execution algorithm of Mercury, an efficient purely declarative logic programming language., *Journal of Logic Programming* **29**: 17–54.

Henderson, F., Somogyi, Z. et Conway, T. (1996). Determinism analysis in the Mercury compiler, *Proceedings of the Nineteenth Australian Computer Science Conference*, Melbourne, Australia, pp. 337–346.

Hermann, M. et Kolaitis, P. G. (1995). Computational complexity of simultaneous elementary AC-matching problems, *in* J. Wiedermann et P. Hájek (eds), *Proceedings 20th International Symposium on Mathematical Foundations of Computer Science, Prague (Czech Republic)*, Vol. 969 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 359–370.

Hintermeier, C., Kirchner, C. et Kirchner, H. (1994). Dynamically-Typed Computations for Order-Sorted Equational Presentations (Extended Abstract), *in* S. Abiteboul et E. Shamir (eds), *Proceedings 21st International Colloquium on Automata, Languages, and Programming*, Vol. 820 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 450–461.

Hintermeier, C., Kirchner, C. et Kirchner, H. (1995). Sort Inheritance for Order-Sorted Equational Presentations, *Recent Trends in Data Types Specification*, Vol. 906 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 319–335.

Hodges, A. (1988). *Alan Turing ou l'énigme de l'intelligence*, Édition Payot. ISBN 2-228-88081-7.

Hoffmann, C. M. et O'Donnell, M. J. (1982a). Pattern-matching in trees, *Journal of the ACM* **29**(1): 68–95.

Hoffmann, C. M. et O'Donnell, M. J. (1982b). Programming with equations, *ACM Transactions on Programming Languages and Systems* **4**(1): 83–112.

Hofstadter, D. (1985). *Gödel, Escher, Bach : les Brins d'une Guirlande Eternelle*, InterÉdition. ISBN 2-7296-0040-X.

Homann, K. et Calmet, J. (1995). Combining Theorem Proving and Symbolic Mathematical Computing, *in* J. C. J. Calmet (ed.), *Proceedings of AISMC-2*, Vol. 814 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 18–29.

Hopcroft, J. E. et Karp, R. M. (1973). An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs, *SIAM Journal of Computing* **2**(4): 225–231.

Hullot, J.-M. (1979). Associative-commutative pattern matching, *Proceedings 9th International Joint Conference on Artificial Intelligence*.

Hullot, J.-M. (1980). *Compilation de Formes Canoniques dans les Théories équationelles*, Thèse de Doctorat de Troisième Cycle, Université de Paris Sud, Orsay (France).

Ishisone, M. et Sawada, T. (1998). Brute: brute force rewriting engine, *Proceedings of the CafeOBJ Symposium'98, Numazu-shi, Shizuoka Prefecture, Japan*, CafeOBJ Project, pp. 1–16.

Jones, R. et Lins, R. (1996). *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*, Wiley. ISBN 0-471-94148-4.

Jouannaud, J.-P., Kirchner, C., Kirchner, H. et Mégrelis, A. (1992). Programming with equalities, subsorts, overloading and parameterization in OBJ, *Journal of Logic Programming* **12**(3): 257–280.

Jouannaud, J.-P. et Kirchner, H. (1986). Completion of a set of rules modulo a set of equations, *SIAM Journal of Computing* **15**(4): 1155–1194. Preliminary version in Proceedings 11th ACM Symposium on Principles of Programming Languages, Salt Lake City (USA), 1984.

Kamperman, J. F. T. (1996). *Compilation of Term Rewriting Systems*, PhD thesis, UVA, Amsterdam, NL.

Kapur, D. et Zhang, H. (1988). RRL: A rewrite rule laboratory, *Proceedings 9th International Conference on Automated Deduction, Argonne (Ill., USA)*, Vol. 310 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 768–769.

Kirchner, H. et Moreau, P.-E. (1995). Prototyping completion with constraints using computational systems, *in* J. Hsiang (ed.), *Proceedings 6th Conference on Rewriting Techniques and Applications, Kaiserslautern (Germany)*, Vol. 914 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 438–443.

Kirchner, H. et Moreau, P.-E. (1996). A reflective extension of Elan, *in* J. Meseguer (ed.), *Proceedings of the first international workshop on rewriting logic*, Vol. 4, Electronic Notes in Theoretical Computer Science, Asilomar (California).

Kirchner, H. et Moreau, P.-E. (1998). Non-deterministic computations in ELAN, *in* J. Fiadeiro (ed.), *Recent Developements in Algebraic Specification Techniques, Proc. 13th WADT'98,*

Selected Papers, Vol. 1589 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 168–182.

Klint, P. (1993). A meta-environment for generating programming environments, *ACM Transactions on Software Engineering and Methodology* **2**: 176–201.

Knuth, D. E. et Bendix, P. B. (1970). Simple word problems in universal algebras, *in* J. Leech (ed.), *Computational Problems in Abstract Algebra*, Pergamon Press, Oxford, pp. 263–297.

Koorn, J. W. C. (1994). *Generating Uniform User-Interfaces for Interactive Programming Environments*, PhD thesis, University of Amsterdam (The Nederlands).

Kounalis, E. et Lugiez, D. (1991). Compilation of pattern matching with associative commutative functions, *16th Colloquium on Trees in Algebra and Programming*, Vol. 493 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 57–73.

Leroy, X. (1995). Le système caml special light: modules et compilation efficace en caml, *Rapport de recherche 2721*, INRIA.

Leroy, X. et Mauny, M. (1993). Dynamics in ML, *Journal of Functional Programming* **3**(4): 431–463.

Lescanne, P. (1983). Computer experiments with the REVE term rewriting systems generator, *Proceedings of 10th ACM Symposium on Principles of Programming Languages*, ACM, pp. 99–108.

Lescanne, P. (1989). Completion procedures as transition rules + control, *in* M. Diaz et F. Orejas (eds), *TAPSOFT'89*, Vol. 351 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 28–41.

Lesk, M. (1975). LEX - a Lexical Analyzer Generator, *CSTR 39*, Bell Laboratories, Murray Hill, N. J.

Lugiez, D. et Moysset, J.-L. (1994). Tree automata help one to solve equational formulae in AC-theories, *Journal of Symbolic Computation* **18**(4): 297–318.

MacMahon, P. A. (1916). *Combinatory Analysis*, Vol. 2, Cambridge University Press, chapter II: A Syzygetic Theory, pp. 111–114. Reprinted by Chelsea, New York, 1960.

Marché, C. (1996). Normalized rewriting: an alternative to rewriting modulo a set of equations, *Journal of Symbolic Computation* **21**(3): 253–288.

McAloon, K. et Tretkoff, C. (1995). 2LP: Linear programming and logic programming, *in* P. Hentenryck et V. Saraswat (eds), *Principles and Practice of Constraint Programming*, MIT Press, pp. 101–116.

McCune, W. W. (1994). Otter 3.0: Reference manual and guide, *Technical Report 6*, Argonne National Laboratory.

Meseguer, J. (1992). Conditional rewriting logic as a unified model of concurrency, *Theoretical Computer Science* **96**(1): 73–155.

Meseguer, J. (1998). Membership algebra as a semantic framework for equational specification, *in* F. Parisi-Presicce (ed.), *Proceedings of WADT'97*, Lecture Notes in Computer Science, Springer-Verlag.

Metzemakers, T. et Sherman, D. J. (1995). Mingus : un compilateur expérimental pour la logique équationnelle, *TR-1052-95*, LaBRI, Université Bordeaux-1, Bordeaux.

Moreau, P.-E. (1994). *Complétion avec contraintes en ELAN*, Rapport de DEA, Université Henri Poincaré – Nancy 1.

Moreau, P.-E. (1998a). A choice-point library for backtrack programming, JICSLP'98 Post-Conference Workshop on Implementation Technologies for Programming Languages based on Logic.

Moreau, P.-E. (1998b). Compiling nondeterministic computations, *Technical Report 98-R-005*, CRIN. **URL:** *file://ftp.loria.fr/pub/loria/protheo/TECHNICAL_REPORTS_1998/Moreau-98-R-005.ps.gz*

Moreau, P.-E. et Kirchner, H. (1997). Compilation Techniques for Associative-Commutative Normalisation, *in* A. Sellink (ed.), *Second International Workshop on the Theory and Practice of Algebraic Specifications*, Electronic Workshops in Computing, eWiC web site: http://ewic.springer.co.uk/, Springer-Verlag, Amsterdam. 12 pages. **URL:** *file://ftp.loria.fr/pub/loria/protheo/COMMUNICATIONS_1997/MoreauK-ASFSDF97.ps.gz*

Moreau, P.-E. et Kirchner, H. (1998). A compiler for rewrite programs in associative-commutative theories, *"Principles of Declarative Programming"*, number 1490 in *Lecture Notes in Computer Science*, Springer-Verlag, pp. 230–249. Report LORIA 98-R-226.

Nakagawa, A., Futatsugi, K., Tomura, S. et Shimizu, T. (1987). Algebraic Specification of Macintosh's QuickDraw Using OBJ2, *Technical Report Draft*, ElectroTechnical Laboratory, Tsukuba Science City, Japan. Proceedings of the 10th International Conference on Software Engineering, Singapore, April 1988.

Nedjah, N. (1997). *Pattern-matching automata for efficient evaluation in equational programming*, PhD thesis, UMIST, Manchester, UK.

Nedjah, N., Walter, C. D. et Eldrige, E. (1997). Optimal left-to-right pattern-matching automata, *in* M. Hanus, J. Heering et K. Meinke (eds), *Proceedings 6th International Conference on Algebraic and Logic Programming, Southampton (UK)*, Vol. 1298 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 273–286.

Ogata, K., Ohara, K. et Futatsugi, K. (1997). TRAM: An abstract machine for order-sorted conditional term rewriting systems, *in* H. Comon (ed.), *Proceedings 8th Conference on Rewriting Techniques and Applications, Sitges (Spain)*, Lecture Notes in Computer Science, Springer-Verlag.

Partington, V. (1997). Implementation of an Imperative Programming Language with Backtracking, *Technical Report P9714*, University of Amsterdam, Programming Research Group. Available by anonymous ftp from ftp.wins.uva.nl, file pub/programming-research/reports/1997/P9712.ps.Z.

Peterson, G. et Stickel, M. E. (1981). Complete sets of reductions for some equational theories, *Journal of the ACM* **28**: 233–264.

Pettersson, M. (1995). *Compiling Natural Semantics*, PhD thesis, University of Linköping, Sweden.

Pottier, L. (1990). Bornes et algorithme de calcul des générateurs des solutions de systèmes diophantiens linéaires, *Technical report*, INRIA Sophia Antipolis.

Sawamura, H. et Takeshima, T. (1985). Recursive unsolvability of determinacy, solvable cases of determinacy and their applications to Prolog optimization, *Proceedings of the Second International Logic Programming Conference*, Boston, Massachusetts, pp. 200–207.

Sekar, R. C., Ramesh, R. et Ramakrishnan, I. V. (1992). Adaptive pattern maching, *in* W. Kuich (ed.), *Proceedings of ICALP 92*, Vol. 623 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 247–260.

Sherman, D. J. (1994). *Run-time and Compile-time Improvements to Equational Programs*, PhD thesis, University of Chicago, USA.

Stallman, R. (1995). *Using and porting the GNU CC compiler.*

Stavridou, V. (1988). Specifying in OBJ, verifying in REVE and some ideas about time, *Technical Report CSD-TR-605*, Department of Computer Science, RHBNC, University of London. To appear in "Experiments with the OBJ Executable Specification Language", D. Coleman, R. M. Gallimore, J. A. Goguen eds.

Strandh, R. I. (1988). *Compiling Equational Programs into Efficient Machine Code*, PhD thesis, The Johns Hopkins University, Baltimore, MD.

Strandh, R. I. (1989). Classes of equational programs that compile into efficient machine code, *in* N. Dershowitz (ed.), *Proceedings of the Third International Conference on Rewriting Techniques and Applications*, Chapel Hill, NC, pp. 449–461. Vol. 355 of *Lecture Notes in Computer Science*, Springer, Berlin.

Turing, A. (1936). On computable numbers, with an application to the entscheidungsproblem, *Proceedings of the London Mathematical Society*, Vol. 42, pp. 230–265.

van den Brand, M. G. J., de Jong, H. A. et Olivier, P. (1998). Efficient annotated terms, *Technical report*, University of Amsterdam. In preparation.

van den Brand, M. G. J., Heering, J. et Klint, P. (1997). Renovation of the Old ASF Meta-Environment – Current State of Affairs, *Proceedings of International Workshop on Theory and Practice of Algebraic Specifications ASF+SDF 97, Amsterdam (The Nederlands)*, Workshops in Computing, Springer-Verlag.

van den Brand, M. G. J., Klint, P. et Olivier, P. (1999). Compilation and Memory Management for ASF+SDF, *Compiler Construction*, Lecture Notes in Computer Science, Springer-Verlag.

van den Brand, M. G. J., Olivier, P., Moonen, L. et Kuipers, T. (1997). Implementation of a Prototype for the New ASF Meta-environment, *Proceedings of International Workshop on Theory and Practice of Algebraic Specifications ASF+SDF 97, Amsterdam (The Nederlands)*, Workshops in Computing, Springer-Verlag.

Vigneron, L. (1998). Automated Deduction Techniques for Studying Rough Algebras, *Fundamenta Informaticae* **33**(1): 85–103.

Visser, E. (1997). *Syntax Definition for Language Prototyping*, PhD thesis, UVA, Amsterdam, NL.

Vittek, M. (1994). ELAN*: Un cadre logique pour le prototypage de langages de programmation avec contraintes*, Thèse de Doctorat d'Université, Université Henri Poincaré – Nancy 1.

Vittek, M. (1996). A compiler for nondeterministic term rewriting systems, *in* H. Ganzinger (ed.), *Proceedings of RTA'96*, Vol. 1103 of *Lecture Notes in Computer Science*, Springer-Verlag, New Brunswick (New Jersey), pp. 154–168.

Wampler, S. et Griswold, R. (1983). The implementation of generators and Goal-Directed Evaluation in Icon, *Software-Practice and Experience* **13**: 495–518.

Warren, D. H. D. (1983). An abstract Prolog instruction set, *Technical Report 309*, SRI International, Artificial Intelligence Center.

Weis, P. et Leroy, X. (1993). *Le langage Caml*, Ediscience. ISBN 2-7296-0493-6.

Wilhelm, R. et Maurer, D. (1994). *Les compilateurs*, Masson. ISBN 2-225-84615-4.

Wilson, P. R. (1992). Uniprocessor garbage collection technique, *International Workshop on Memory Management*, Vol. 637 of *Lecture Notes in Computer Science*, Springer-Verlag, Saint Malo, pp. 1–42.

Wirsing, M. (1995). Algebraic specification languages: An overview, *Recent Trends in Data Types Specification*, Vol. 906 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 81–115.

Zendra, O., Colnet, D. et Coucaud, P. (1998). With SmallEiffel, The GNU Eiffel Compiler, Eiffel joins the Free Software community., *GNU Bulletin* **25**. To be published.

Index

AC, *voir* associatif-commutatif
accès, fonction d', 94
algorithme, d'Albert Gräf, 79 ; avancé de setChoicePoint et fail, 106 ; incrémental de construction d'arbre, 67
alias, d'un opérateur, 12
Alma, 101
alphabet, 62
analyse, du déterminisme, 131 ; lexicale, syntaxique, sémantique, 29
application, d'une stratégie, 16 ; d'une substitution, 14 ; de taille « réelle », 175
approche, hybride d'ELAN, 147 ; la nouveauté de notre, 57
arbre, de filtrage, 67
architecture, du compilateur, 143
argument, `@`, 12
arité, $\#s$, 62 ; d'une fonction, 13
arrêt, d'une machine de Turing, 16
artificielle, intelligence, xiii
ASF+SDF, 2 ; présentation, 35
associatif, *list-matching*, 45
associatif droite, `assocRight`, 12
associatif gauche, `assocLeft`, 12
associatif-commutatif, classe de motifs, 88 ; compilation du filtrage, 81 ; niveau d'un symbole, 88 ; présentation, 21 ; théorie, 82
associativité, équation d', 82
`@`, argument, 12 ; opérateur d'injection, 12, 157
automate, à mémoire, 72 ; bloquage d'un, 73 ; canonique, 65 ; d'arbre, 175 ; de filtrage, 63 ; de filtrage avec jumpNode, 75 ; déterministe, 62 ; faiblement canonique, 71 ; faiblement déterministe, 65 ; non déterministe, 65
avantage, d'un compilateur, 32

backtrack, 100
bande, de lecture, 65, 73
benchmark, 165
BG, *voir* graphe biparti
bibliothèque, d'ELAN, 27 ; taille de la, 155
bloquage, choix responsable d'un, 76 ; d'un automate, 73
bool3, programme, 170
Borovanský, P., thèse de, 29
builtin, *voir* élémentaire

CafeOBJ, 1 ; présentation, 35
Caml, 1, 162
canonique, automate, 65 ; automate faiblement, 71 ; construction d'un terme, 125 ; terme en forme, 82
caractéristique, d'ELAN, 21
CBG, *voir* graphe biparti
Cg, 102
chaîne, bien formée, 62 ; longueur d'une, 62 ; terme vu comme une, 62
chevalier, xiii
choix, du langage d'implantation, 57 ; méta, 58 ; point de, 99 ; responsable d'un bloquage, 76
choose/try, présentation, 20
cible, langage, 54, 143
C*i*ME, 1
Claire, 101
classe, de motifs, 88 ; héritage des, 147
classification, du déterminisme, 132
Clean, 1
clos, terme, 13
clôture, calcul incrémental, 69 ; d'un ensemble, 65, 66 ; ∇, 67 ; réduite, 70
codomaine, d'une fonction, 13
colette, programme, 175
communication, avec le monde extérieur, 43 ; outil de, 40
commutativité, équation d', 82
comparaison, avec d'autres implantations, 178 ; d'environnements, 33 ; des approches, 78
compilateur, architecture du, 143 ; avantage d'un, 32 ; d'ELAN, 32 ; définition d'un, 53 ; fonctionnement du, 150 ; objectif d'un, 32 ; or-

ganisation du, 147 ; prototype de, 57 ; taille du, 150
compilation, de la réécriture, 58 ; degré de, 166 ; des conditions, 123 ; des évaluations locales, 118 ; des fonctions d'accès, 94 ; des règles et des stratégies, 113 ; des stratégies, 127 ; du filtrage, 116 ; du filtrage associatif-commutatif, 81 ; du filtrage syntaxique, 61 ; du processus de normalisation, 127 ; évaluation des méthodes, 165 ; modulaire, 143 ; séparée, 146
complétion, de Knuth Bendix, 14, 170
compteur, de références, 159
concaténation, d'un élément à une liste, 17 ; de stratégies, 17, 127
conception, d'ELAN, 39 ; méta, 53
condition, compilation des, 123 ; présentation, 19 ; règle avec, 15
confluence, d'un système, 14
conservatif, ramasse miettes, 161
construction, d'une substitution, 94 ; du terme réduit, 124
contrôle, de la sélection, 132 ; du nombre de résultats, 132
coopération, avec Mark van den Brand, 46 ; ELAN ASF+SDF, 45
copie, ramasse miettes avec, 160
correction, des spécifications, 27
couche, supérieure syntaxique, 83
couleur, d'un terme, 127
coût, du filtrage AC, 176
création, d'outils, 43
cut, 116

daTac, 1
décision, procédure de, 63
décodage, des instructions, 54
degré, de compilation, 166
déplacement, de la tête de lecture, 73
destructive update, 124
détection, d'erreur, 136
déterminisme, analyse du, 131 ; classification du, 132 ; gestion du, 99 ; impact de l'analyse du, 136 ; inférence du, 134 ; mode de, 133 ; uniforme, 123
déterministe, automate de filtrage, 62 ; stratégie, 133
d-mode, 133 ; d'une règle, 135 ; inférence du, 134
domaine, d'une fonction, 13
dont care choose, définition de, 17
dont care one, définition de, 17
dont know choose, définition de, 17

échec, fail, 99 ; situation d', 67
égalité, $=_{AC}$, 22
Eiffel, approche hybride, 147
ELAN, 2 ; formalisme de spécification, 11 ; mini, 139, 174
élémentaire, module, 27 ; sortes et opérations, 156
emballage, de termes, 157
ensemble, clôture d'un, 65, 66 ; clôture réduite d'un, 70 ; de motifs, 63 ; de termes, 13 ; de variables, 13 ; fini d'états, 63
environnement, comparaison des, 33 ; de spécification, 27
ϵ, position vide, 13
équation, diophantienne, 84
Equational Logic Programming, 2
Erlang, 102
erreur, détection d', 136
état, ensemble fini, 63 ; final, 63 ; initial, 63
étiquette, d'une règle, 16
évaluation, des méthodes de compilation, 165 ; des performances, 172
évaluation locale, compilation des, 118 ; `Local-Evaluation`, 148 ; présentation, 19 ; puissance des, 121
expérimentation, résultat, 138
exploration, avec une stratégie, 127
expressivité, de la réécriture, 23
extension, de la classe des motifs, 95 ; du langage de stratégie, 28 ; variable d', 88
extraction, d'un graphe biparti, 90

factorielle, en ELAN, 119
faiblement canonique, automate, 71
faiblement déterministe, automate, 65
fail, 99
Fibonacci, 166 ; programme déterministe, 139
filtrage, 19 ; arbre de, 67 ; associatif, 45 ; automate de, 63 ; compilation du, 116 ; compilation du filtrage associatif-commutatif, 81 ; compilation du filtrage syntaxique, 61 ; coût du filtrage AC, 176 ; *many-to-one*, 61 ; *one-to-one*, 61 ; problème de, 83 ; procédure

AC *many-to-one*, 84 ; procédure AC *one-to-one*, 83 ; structure AC, 84 ; sur les mots, 61
filtre, 14
first, définition de, 17
first one, définition de, 17
fonction, arité d'une, 13 ; codomaine d'une, 13 ; d'accès, 94 ; domaine d'une, 13 ; factorielle en ELAN, 15 ; factorielle en ELAN, 15, 119 ; profil d'une, 13
fonctionnement, du compilateur, 150
formalisme, ATerms, 46 ; ELAN, 11
format, asFix, 46 ; d'échange, 35, 40 ; Efix, 46 ; REF, 40
forme, aplatie, 31 ; canonique, 82 ; normale, 14
Futatsugi, K., 2

garbage collector, *voir* ramasse miettes
générateur, 120
génération, modulaire, 145 ; ramasse miettes à, 162
gestion, de la mémoire, 32, 158 ; de déterminisme, 99
glouton, rafinement, 93
GNU, C, 102 ; Eiffel, 147
Gödel, K., 1
grammaire, hors contexte, 11 ; signature, 11
graphe biparti, 83 ; compact, 88 ; représentation d'un, 154

Heering, J., 2
héritage, des classes, 147
hybride, approche d'ELAN, 147 ; ramasse miettes, 162

Icon, 102
impact, de l'analyse du déterminisme, 136 ; sur la sélection, 136
implantation, d'un langage, 54 ; de l'interpréteur, 31 ; de setChoicePoint et fail, 103 ; détaillée, 107 ; par Marian Vittek, 56, 159 ; par Steven Eker, 31
incrémental, algorithme de construction d'arbre, 67 ; calcul d'une clôture, 69
index, moi-même, 245
inefficacité, de l'interpréteur, 31
inférence, du déterminisme, 134 ; du d-mode, 134
informatique, xiii
injection, symbole d', 157
instance, réductible, 126
instanciation, d'une variable, 124
intégration, d'un composant, 47 ; du compilateur, 40
intelligence, artificielle, xiii
intérêt, des symboles AC, 23
interpréteur, d'ELAN, 30 ; définition d'un, 53 ; inefficacité de l', 31

Janus, 102
Jaoui, A., xiii
Jouannaud, J.-P., 2
jumpNode, 71 ; automate de filtrage avec, 75

KL1, 102
Klint, P., 2, 35

label, d'une règle, 16
lac, de Paladru, xiii
langage, assembleur, 15, 99 ; C, 102 ; cible, 54, 143 ; d'implantation, 57, 143 ; de haut niveau, 15 ; de spécification, 11 ; esprit du, 18 ; façon d'implanter un, 54 ; impératif, 109 ; machine, 55 ; méta langage de stratégie, 30 ; micro code, 55 ; portable, 109 ; source, 54, 143
Larch Prover, 1
lecture, bande de, 65, 73 ; tête de, 63
leftmost-innermost, 16
leftmost-outermost, 16
lexème, `Lexem`, 149
lexicale, analyse, 29
lien, vers lepère, 76
linéaire, terme semi, 83
liste, module en ELAN, 17 ; paramétré en ELAN, 24
longueur, d'une chaîne, 62

machine abstraite, 45 ; définition d'une, 53
machine de Turing, arrêt d'une, 16
many-to-one, approche, 84 ; filtrage, 61
marquage, ramasse miettes avec, 160
mathématique, xiii
Maude, 2 ; présentation, 33
McCarthy, J., 1
membre gauche, réutilisation du, 124
mémoire, automate à, 72 ; gestion de la, 32, 158
Mercury, 102

méta, choix, 58 ; conception, 53 ; environnement, 35 ; langage de stratégie, 30
minela, 174
mini, ELAN, 139, 174
ML, 1
mode, de déterminisme, 133
modularité, 23 ; de la compilation, 143 ; du *parsing*, 144
module, *builtin*, 27 ; liste en ELAN, 17 ; paramétré, 24 ; réorganisation des, 145
modulo AC, égalité, 22
moi-même, 247
mot, clé `if`, 15 ; filtrage sur un, 61
motif, classe de, 88 ; ensemble de, 63 ; extension à l'ensemble des, 95 ; initialisation des listes de, 90
multi-résultats, stratégie, 133
multiplicité, d'un terme, 83

∇, 67
nat10, programme, 173
niveau, d'un symbole AC, 88
nom, d'une règle, 16 ; opérateur sans, 12
déterminisme, classification du, 132
non-déterminisme, analyse du, 131 ; gestion du, 99 ; uniforme, 123
non-déterministe, stratégie, 133

O'Donnell, M. J., 1
OBJ, 2 ; présentation, 33
objectif, d'un compilateur, 32 ; de cette thèse, 23 ; des spécifications algébriques, 27 ; du groupe ELAN, 57
one-to-one, approche, 83 ; filtrage, 61
opérateur, *And*, 133 ; *builtin*, 28 ; d'injection, 12 ; *Or*, 133
opération, élémentaire, 156 ; prédéfinie, 155
ordonnancement, 113
ordre, sur les d-mode, 133
organisation, du compilateur, 147
originalité, d'ELAN, 16, 36 ; de Maude, 34 ; du préprocesseur, 29
Otter, 1
outil, création d', 43 ; de communication, 40 ; pour spécifier, 27

Paladru, lac de, xiii
parallel-innermost, 16
parallel-outermost, 16
parseur, d'ELAN, 29, 40 ; de termes infixés, 48 ; modulaire, 144 ; `REFParseur`, 148
pattern matching, *voir* filtrage
père, lien vers, 76
performance, évaluation des, 172 ; problème de, 32
polynôme, exemple en ELAN, 22
position, dans un terme, 13 ; vide, 13
Post, E., 1
préfixe, d'un terme, 66 ; recouvrement de, 70
préprocesseur, d'ELAN, 29
preuve, de propriété de programme, xiv
primitive, de gestion des points de choix, 101, 115 ; de stratégies, 132
priorité, d'un opérateur, 12
problème, de filtrage AC, 83 ; *de reshuffling*, 145 ; des n reines, 119 ; lié à la compilation modulaire, 143 ; lié à la récursivité, 136
procédure, de décision, 63 ; de filtrage AC *many-to-one*, 84 ; de filtrage AC *one-to-one*, 83
profil, d'une fonction, 13
programmation, impérative, 109
programme, ANS-Complétion, 170 ; bool3, 170 ; colette, 175 ; fib, 139 ; fib_builtin, 166 ; minela, 139, 174 ; nat10, 173 ; nqueensAC, 167 ; p5, 138 ; queens, 139 ; set, 173 ; somme, 176
Prolog, cut, 116
portabilité, 109
prototype, de compilateur, 57

qualité, des spécifications, 23, 24
`query2ref`, 44

racine, d'un terme, 16
raffinement, glouton, 93
ramasse miettes, 159 ; à génération, 162 ; avec compteur de références, 159 ; avec copie, 160 ; avec marquage, 160
recherche, stratégie de, 73
recouvrement, de préfixes, 70
récursivité, dans l'analyse du déterminisme, 136
ReDuX, 1
réécriture, compilation de la, 58 ; conditionnelle, 15 ; système de, 13
référence, compteur de, 159
réflexivité, de Maude, 34
règle, compilation des, 113 ; conditionnelle, 15 ; de transition d'états, 63 ; d-mode d'une, 135 ;

et stratégie, 16 ; gloutonne, 93 ; impact sur la sélection d'une, 136 ; nommée, 16 ; `RewriteRule`, 148 ; sélection d'une, 116
reine, impact de l'analyse du déterminisme, 139 ; problèmes des, 119 ; programme nqueensAC, 167
remplacement, d'un sous-terme, 13
renormalisation, des instances réductibles, 126
réorganisation, des modules, 145
répétition, stratégie de, 129
représentation, aplatie, 40 ; des graphes bipartis compacts, 154 ; des termes, 153 ; des vecteurs de bits, 154
`ref2result`, 44
Resnay, A., xiii
résultat, à la demande, 18 ; contrôle du nombre de, 132 ; expérimentaux, 138 ; extraction des, 17 ; tous les, 17 ; un seul, 17
retour arrière, 100
réutilisation, du membre gauche, 124
Reve, 1
RML, 102
RRL, 1

sélection, contrôle de la, 132 ; d'une règle, 19, 116 ; impact sur la, 136
sémantique, analyse, 29
semi-compilation, 54
semi-déterministe, stratégie, 133
semi-linéaire, terme, 83
set, programme, 173
setChoicePoint, 99
Σ, signature, 13
σ, substitution, 14
$\sigma(t)$, application d'une substitution, 14
signature, 11, 13 ; grammaire, 11 ; Σ, 13 ; syntaxe d'une, 11
situation, d'échec, 67
solution, calcul d'une seule solution AC, 93 ; existence d'une, 86
somme, programme, 176
sorte, 11 ; *builtin*, 28 ; élémentaire, 156 ; injectée, 13
source, langage, 54, 143
sous-terme, 13
spécification, algébrique, 11 ; environemment de, 27 ; langage de, 11
Spike, 1
stratégie, det, semi, multi, nondet, 133 ; application d'une, 16 ; compilation des, 113, 127 ; d'application, 15 ; d'exploration, 127 ; de concaténation, 17, 127 ; de normalisation, 16 ; de recherche, 73 ; de répétition, 129 ; définie par l'utilisateur, 16 ; *dont care choose*, 17 ; *dont care one*, 17 ; *dont know choose*, 17 ; élémentaire, 16 ; *fail*, 17 ; *first*, 17 ; *first one*, 17 ; identité, 17 ; impact sur la compilation d'une, 137 ; *iterate*, 17 ; *leftmost-innermost*, 16 ; *leftmost-outermost*, 16 ; *parallel-innermost*, 16 ; *parallel-outermost*, 16 ; primitive, 132 ; *repeat*, 17 ; sélection d'une, 17 ; `StrategyTerm`, 148
structure, compacte de graphe biparti, 88 ; de donnée, 153 ; de filtrage AC, 84
substitution, 14 ; calcul d'une, 94 ; construction d'une, 94
$t\sigma$, application d'une substitution, 14
suffixe, ajout d'un, 68 ; d'un terme, 66
sujet, 14 ; radical, 14
sûreté, des spécifications, 23
symbole, associatif-commutatif, 21 ; chaîne de, 62 ; constructeur, 59 ; d'injection, 157 ; défini, 59 ; niveau d'un symbole AC, 88 ; `Symbol`, 149
syntaxe, d'une signature, 11
syntaxique, analyse, 29 ; compilation du filtrage, 61 ; couche supérieure, 83 ; théorie, 61
système, confluent, 14 ; de réécriture, 13 ; ouvert, 45 ; terminant, 14

$\mathcal{T}_s(\mathcal{F},\mathcal{X})$, ensemble de termes, 13
taille, de la bibliothèque, 155 ; du compilateur, 150
technique, d'indexage, 62 ; hybride, 56
terme, annoté, 46 ; aplati, 31, 82 ; avec multiplicité, 83 ; bien formé, 62 ; clos, 13 ; coloré, 127 ; $\hat{t}$, 83 ; emballé, 157 ; en forme canonique, 82, 125 ; ensemble de, 13 ; `Flatterm`, 149 ; irréductible, 14 ; position dans un, 13 ; préfixe d'un, 66 ; réduit, 124 ; $t[t']_\omega$, 13 ; représentation d'un, 153 ; requête, 30 ; semi-linéaire, 83 ; sommet d'un, 16 ; $t_{|\omega}$, 13 ; suffixe d'un, 66 ; sujet, 14 ; `Terme`, 148 ; vu comme une chaîne, 62
terminaison, d'un système, 14 ; d'une machine de Turing, 16

tête de lecture, 63 ; déplacement de la, 73
théorie, associative-commutative, 82 ; syntaxique, 61
thèse, xiii ; de Marian Vittek, 11 ; de Peter Borovanský, 29
transformation, de règles, 95
transition, règle de transition d'états, 63
Turing, A., 1
PE machine de, 16

utilisateur, stratégie définie par l', 16

van den Brand, M. G. J., visite à Nancy, 46
variable, d'extension, 88 ; ensemble de, 13 ; instanciation d'une, 124 ; instanciation des variables apparaissant sous un symbole AC, 94 ; locale, 105
Vittek, M., compilateur ELAN, 57, 159 ; interpréteur ELAN, 56 ; thèse de, 11

where, présentation, 19

$\mathcal{X}_s$, ensemble de variables, 13